

Recent developments in and around B2000¹

Silvio Merazzi, Mathias Doreille,
Thomas Ludwig, Pieter Volgers

SMR SA
P.O. Box 4014
CH-2500 Bienne 4

Since the last *B2000* workshop in Twente, The Netherlands, two years ago, substantial development of the *B2000* system has been brought about by SMR. On the one hand new technologies have been explored, on the other the capabilities of existing parts of *B2000* have been further improved. In addition to the development of the *B2000* system, SMR has continued to assure and improve maintenance of the system, one of SMR's main tasks. This paper is intended to illustrate some of the developments and it also serves as a basis for a discussion regarding future directions.

1. The B2000 base code

Configuration

The *B2000* code configuration has been improved. Newer versions of *autoconf*, *automake* and *libtool* had to be installed and the make files had to be updated. These new versions had many bug fixes. Unfortunately the last stable versions of the tools still contain bugs and missing functionality. *automake* had to be patched in order to add *Fortran* dependencies. A development version of *libtool* extracted from the CVS tree of *libtool* had to be selected which supports properly shared *Fortran* libraries. The packages are on the *SMR* server.

B2000 processors

The use of *B2000* at the *DLR* Stuttgart helped in the continued development of the explicit dynamic analysis processor, *b2eta*. Apart from the developments at the *DLR*, *SMR* improved the code by:

- Introducing self-contact. Some debugging and improvements are still needed.
- New implementation of the material classes, which facilitates the introduction of new material models. Any element making use of this new scheme will have direct access to the implemented material model (if applicable).
- New implementation or re-implementation of the following material models:
 - Orthotropic elastic material.
 - Layered elements (layer-wise integration or using layer-theory).
 - *Linköping* rate-dependent UD damage model.
 - *DLR* fabric damage model.
- Improvement of the existing shell element Q4 . ETA.
- Introduction of a shell element using 2x2 integration (to be completed).

¹ Presented at the 4th B2000 workshop, 24-25.10.2002, Bienne, Switzerland.

DLR Stuttgart is planning to implement the standard Belytschko shell element, which is, although not very accurate, by far the most robust shell element available. Furthermore, *SMR* will modify the existing 8-node volume element to make use of the new material scheme and new element implementations are planned.

The implicit symmetric sparse solver has been integrated in *B2000* in the macro-processor *b2slin*. This processor will be, in the future, a complete replacement of the macro-processor *b2lin*. For this *b2slin* takes the same set of arguments than *b2lin*. Moreover, an eigenvalue solver based on the symmetric sparse solver has been implemented and will be integrated shortly in the *b2slin* macro processor. After that, the *b2slin* macro-processor can perform linear static and dynamic analysis and will be a nearly complete replacement of *b2lin*. Only elements devised by Gert Rebel and elements that generate singular stiffness matrix do not work correctly with the *b2slin* macro-processor.

A continuation algorithm based on the symmetric sparse solver has been developed. For this, the symmetric sparse solver has been modified to support the *Felippa* technique. With this technique, the continuation method can pass through a simple limit point without failure. This algorithm will be properly integrated in *B2000* in the macro-processor *b2scont*. It has been in use for years in the implicit contact processor.

The contact processor consists of two parts. (1) The detection of the contact and (2) the computation of the contact forces. Both parts are equally important and time consuming. The computation of the contact forces makes use of the continuation method described above. For the detection of the contact an efficient method of detecting the nodes in contact can greatly reduce the computation time. *SMR* has developed an efficient implementation of the contact search for its explicit solver, which could be used with some minor modification for the implicit solver.

B2000 Library

The gradient storage scheme has been revised. The file structure allows for a more consistent description of gradients and their storage scheme: Up to now the gradient components were not consistent throughout the element library. Thus, while one specific element had component 1 defined as σ_{xx} , another one had component 6 defined as σ_{xx} . This was not a problem as long as results were not processed further, like in a viewer, where the specification of a specific component would lead to wrong selections in case different elements with different component definitions were selected! From version 2.5 and up, gradient column names can be defined in the element parameter tables (see paragraph below). Specifically, the description of the position and order of gradient evaluation points have been adapted to the new gradient storage scheme. Thus, for Q (quadrilateral) 2D elements and 2 by 2 Gauss evaluation the gradient scheme description is defined as GRADGAUSS2x2 (see include file *b2constants.ins* for a list of all definitions). Likewise, for HE (hexahedral) 3D elements and 2 by 2 by 2 Gauss evaluation the scheme is GRADGAUSS2x2x2 and for 1D elements the scheme is GRADGAUSS2x1. Most element gradient calculations have already been adapted to the new scheme.

A *B2000* simulation is now described by a directory containing most of the information. The directory is called the *B2000* model directory and the directory name is composed of the model name given by the user and the suffix *.b2m* which is automatically added by the system. By specifying the *B2000* model name, any *B2000* processor tries to create or open the database files contained under the *B2000* model directory. The old style commands *adb*, *cdb*, etc. are still functioning, but they are not supported nor documented anymore.

B2000 database

The definition of the element parameters on the database has been redesigned. The element parameters of a model are now stored in the data set *ELEMENT-PARAMETERS*. It replaces data sets *EPAR* and *ENAM*. Databases with the old tables are still valid. All relevant code has been updated or modified.

The B2000 documentation

The presentation of the *B2000* documentation has been completely revised. The decision has been made to migrate to a true XML (Extensible Markup Language) documentation system, the *DocBook* system (see <http://www.docbook.org>). *DocBook* offers a free DTD (Document Type Definition) and many XSLT style sheets (XML Style sheet Language Transformation). The DTD specifies how to structure the document with markups (chapters, sections, paragraph,s etc). The style sheet is a specification for a XSLT processor on how to convert this structured document to another format.

The *DocBook* DTD and a set of XSLT style sheets are developed and supported by O'REILLY and are freely available. There are actually XSLT style sheets to convert the document to the following formats: HTML, XHTML, man page, info, PDF and RTF.

The XSLT processor is a standard XML tool (<http://www.w3.org/Style/XSL>), not only used by *DocBook*. Thus many free and commercial implementations has available. We chose to use the *libxslt* implementation that is LGPL and written in C (more efficient than the Java implementation).

For the PDF output format a specific XSLT processor called a XSTL-FO processor is needed. This processor is more difficult to implement and actually only commercial products give good results. However, free XSLT_FO processors (apache FOP, PassiveTex, XSLFO) are actively developed. FOP can already compete with commercial products for not too complex XML input. In order not to push *B2000* users to buy a commercial product to generate PDF documentation we temporarily chose to switch to SGML (the precursor of XML) and use the free SGML processor *OpenJade* based on *Tex* to generate the PDF files.

All *B2000* and *b2test* documents have been converted from *SML* and *FrameMaker* to *DocBook* by means of scripts and by manual intervention. *DocBook* and *XML* tools are under active development and the resulting HTML and PDF documents are not completely perfect yet. However, we hope that we have pushed the *B2000* documentation in the right direction.

2. A new development and execution framework for B2000

A framework for replacing the current *B2000* subroutine and function library is still under development. Choosing *Python* as the new *B2000* scripting language does not really have an direct impact on the specification of the new *B2000* class library. A four layer model as it is already implemented in *baspl++* has been selected:

- *The Graphical User Interface layer*
The Graphical User Interface is built with the *GLADE* Graphical User Interface builder. *GLADE* generates *GTK+* library calls. Callbacks (GUI events) are processed by *Python* or by direct connection to the *C++/Fortran* libraries.
- *The scripting User Interface layer*
The choice has been brought to *Python*. *Python* is simple, powerful, extensible. With *Python*, the *B2000* user interface looks consistent and uniform.
- *Wrapping layer*
Wrapping of *Fortran* subroutines and commons is achieved by means of *f2py* (see <http://pyfortran.sf.net>), and wrapping of *C++* with *SWIG* (see <http://www.swig.org>). This technology is in the process of rapid evolution and the tools may change in the future.
- *Libraries*
C++ class libraries and *Fortran* libraries (legacy code, like elements).

An experimental *Python* module *pyb2000* has been built. It is primarily meant as a replacement of the *PCL* commands, the *Fortran* based *PCL* library, and the *B2000* processors (*Fortran* 'main' programs, and the macro-processors. *pyb2000* is a simple *Python* class implemented in the form of a *Python* module. It contains methods that call the *B2000* control modules (*Fortran* subroutines) as well as some utility

subroutines. It also writes directly into some *B2000* commons (nasty, but inevitable). The tool for wrapping *Fortran* is the Fortran to Python Interface Generator (or *f2py*), a public package developed by P. Peterson, see <http://cens.ioc.ee/projects/f2py2e>. An example of running a linear analysis is given below, together with the *PCL* counterpart.

PCL	pyb2000
<pre>B2ip >b2ip.out >>/ opendb demo input demo.mdl go / b2lin >b2lin.out <</ opendb demo go /</pre>	<pre>import b2000 # Create a model database m=b2000.model("demo") # Run the input processor with MDL file demo.mdl m.ip("demo.mdl") # Run the linear analysis processor m.lin()</pre>

The new concept is very well suited for integrating in a GUI. However, there are some fundamental differences to the *B2000* controller presented at the last *B2000* workshop in 2000:

- *B2000* processors, like *b2ip* or *b2lin*, are not launched as sub-processes. They are integral parts of the *pymemcom* module.
- The *B2000* library is now dynamically linked by means of a *Python* module.
- *Tcl/tk* is not used anymore, neither for scripts nor for graphics.

3. The B2000 viewers

The current version of *baspl++*, i.e 1.0.4, has been extensively tested and has reached a stable state. It is now based on the following technologies:

- *The Graphical User Interface*
Implemented with the *GTK+* toolkit and *Python*.
- *The scripting User Interface*
A *Python* module.
- *The core of baspl++*
Implemented in *C++*.

The graphical rendering of *baspl++* has been optimised for off-the-shelf graphics controllers like those found PC's. As an example of a large CFD model the structured multi-block CFD mesh (*NSMB3* solver) of an aeroplane is processed with *baspl++*. The mesh parameters are listed below:

```
Number of blocks:      162
Number of nodes:      7593712
Number of elements:   7031888
```

The time to load the complete mesh and to extract the surfaces defining the aeroplane is approximately 12 seconds on a Pentium 4 PC with 1GB RAM and SCSI disks.

Many improvements have been made and new features have been added to *baspl++*. Some of them, i.e the interactive cutting box and the streamline module are described.

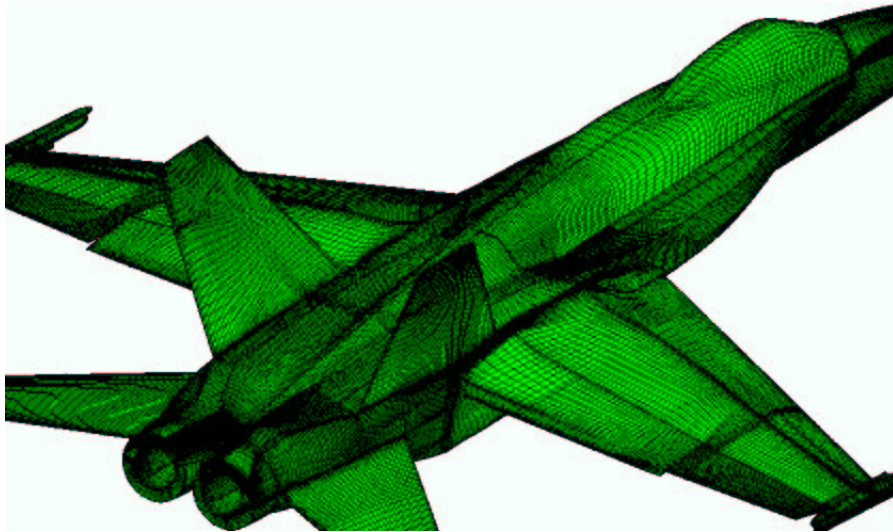


Illustration 1 Structured multi-block CFD mesh (surface extraction)

Interactive plane cutting

At the beginning of this year, *baspl++* still lacked an interactive cutting feature, as it was available in the old *baspl*. This feature is now integrated in *baspl++*, supporting cuts through all linear element types. Higher order elements are “linearised”, i.e cuts are calculated using the edge vertices only. The cutting function is normally applied to every element in the model. Thus, for large models with many elements a good performance is desirable. The cutting feature provides a “fast” option which makes use of advanced algorithms to sort out elements that are in the vicinity of the cutting plane. This option dramatically improves the performance on large models (especially on structured meshes), with computing times on an off-the-shelf Linux PC of a fraction of a second for meshes with approximately 10 million elements.

Cuts are represented like *Parts*, except that they contain derived quantities (i.e interpolated values). All display methods that are available for *Parts*, such as deformed geometry, vector display, colouring and texturing of scalar values etc., can be applied to cuts. Also, if fields are selected and the cutting plane is modified afterward, the values of those fields are automatically re-interpolated. This simplifies the use of the cutting feature and the writing of scripts. The relevant code in *baspl++* have been adapted to provide this functionality.

Cuts can be created by issuing the proper *Python* commands: A cut object is created by supplying the origin and normal vectors of the cutting plane. However, it requires some calculations by hand to find out the correct values for these vectors. Therefore an interactive cutting tool (the *Cut Editor*) has been devised, which allows for interactively placing the cutting plane. This editor is integrated in the *Cut* class and will pop up whenever the origin and normal vectors are not specified. By moving the sliders or by typing in the exact values, the cutting plane can be moved in any direction. A bounding box shows the dimensions of the model. The cutting plane can be drawn symbolically as shown in the figure below, or the cut can be displayed “live”, showing the interpolated mesh and the quantities. The “live” display feature is particularly useful for cutting complicated three-dimensional models, as it helps to understand by “traveling” through such models.

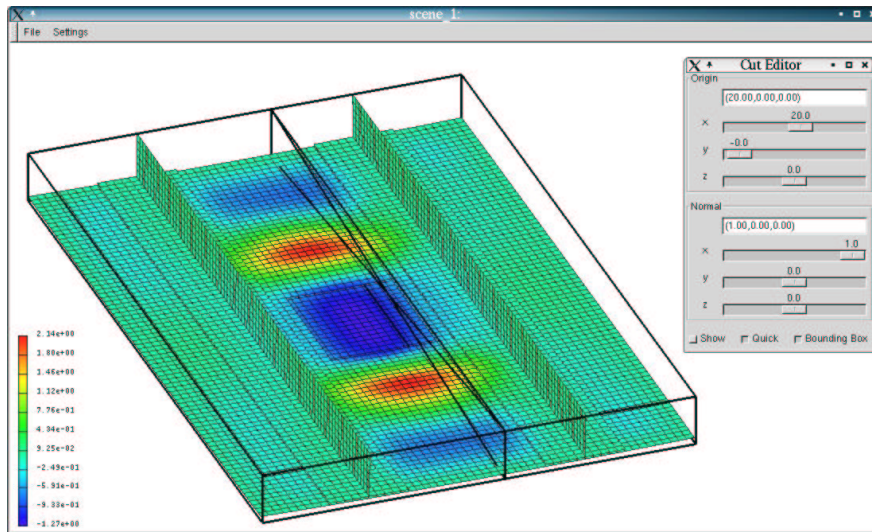


Illustration 2 *baspl++* cutter tool

Streamlines

Streamlines are of interest for CFD applications, although one could see applications in CSM. To implement streamlines one needs three ingredients:

- Find the element enclosing the current position.
- Determine the local element coordinates of this point.
- Using the velocity field, calculate the displacement of the point in time (this is done with a Runge-Kutta integration).

Unlike other streamline implementations, *baspl++* does not divide pyramidal and hexagonal elements into tetrahedral elements. We have implemented an object-oriented framework which contains the element-specific parts of the operations mentioned above. Thus, new element types can be added without modifications of the other parts of the code. Furthermore, great effort has been invested to implement these operations as efficiently as possible. The net result is better accuracy at high performance.

The streamlines feature is currently provided as a plug-in. It is planned to integrate the module in *baspl++*, as it has been done for cuts. It is also planned to support two-dimensional streamlines on surface elements. Figure 3 views some streamlines calculated on an mesh with over 7 million cells.

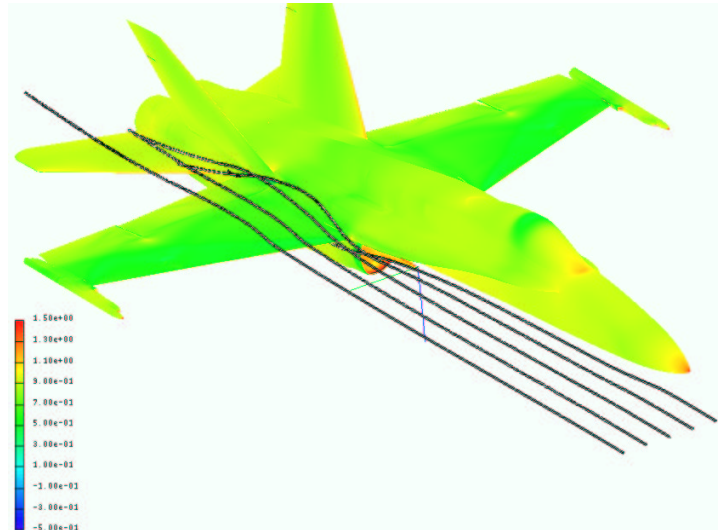


Illustration 3 Calculation and viewing of streamlines

A Graph viewer

The new graph viewer is integrated in *baspl++* and it replaces the *b2xy* program. It is accessed by mean of the experimental *Python* classes *Graph* and *Curve*. A GUI for defining and manipulating graphs is in preparation. The table below demonstrates the new syntax (right column) and it also contains the old *b2xy* syntax for comparison. Note that, since the new graph viewer is integrated in *baspl++*, the viewing system is much more consistent than the old combination of *baspl/baspl++* and *b2xy*.

Old b2xy syntax	New Graph tool syntax
<pre>open test.b2m/archives.mc selgraph clear graph 1 axis x field GLVA cycle 1:100 case 0 key TIME axis y field DISP cycle 1:100 case 0 node 4 comp 3 symbol 1 endgraph endselgraph</pre>	<pre>m = Model('test.b2m') set = m.subset(cycle=(1,100)) x=set.subset(name='GLVA', key='TIME') y=set.subset(name='DISP', node=4, comp=3) c = Curve(x,y) g = Graph() g.add(c) s = Scene() s.add(g)</pre> <div data-bbox="965 1417 1270 1722" style="text-align: center;"> </div>

4. MemCom

The current version of *MemCom* is 7.1, which is a wrap-up of the patches of 7.0. A number of errors have been corrected, new features have been added, and the DB and TB systems have been optimised. The most notable new features are sliced access to positional datasets and the complete re-implementation of the client/server code. Sliced dataset access is interesting for extraction from multi-dimensional datasets (example: Extract every third column of a two-dimensional dataset). The client/server feature has been integrated seamlessly in *MemCom* and is enabled by starting the *MemCom* server program (*mcServer*) on the host where the database is located and then specifying this host in the call to *mcOpen*. Thus all programs that have been programmed to use *MemCom* locally can also use *MemCom* in client/server mode.

The *MemCom* server makes use of transactions to provide concurrent access while maintaining database integrity. For performance reasons not all transaction properties provided by relational database management systems are available. Two new calls (*mcBegin* and *mcEnd*) have been added to make use of the transaction feature.

The *MemCom* server has been implemented as efficiently as possible while maintaining portability. It runs on all *UNIX* systems and has excellent performance.

The *MemCom* data browser contains a number of new and handy features. One of them is the of the directory: In case many data sets are present the directory list becomes very long, as can be seen in the example directory below, which contains over 20000 sets. By selecting the 'tree' representation in the Options menu the directory becomes more readable.

Name	Type	Faddr	Dim1	Dim2
ADIR	I	0	1500	
ANGLE	F	583528944	2700	1
CD	F	583629056	2050	1
CDES	F	583524144	300	1
CF_1	\$	139872240	8192	
CF_2	\$	361695952	8192	
CF_3	F	140126784	144	3
CF_4	F	361960688	144	3
CF_5	F	140155664	144	3
CF_6	F	361987568	144	3
CH_1	F	140182544	84	3
CH_2	F	362014448	84	3
CH_3	F	140198224	144	3
CH_4	F	362030128	144	3
CH_5	F	140238544	144	3
CH_6	F	362070448	144	3
CL	F	140265424	144	3
CM	F	362097328	144	3
COOR	F	140292304	144	3
-1	F	1236384	27000	1
-2	F	1460576	213750	1
-3	F	3178768	213750	1
CF_1.39.1	F	140126784	144	3
CF_1.39.2	F	361960688	144	3
CF_1.43.1	F	140155664	144	3
CF_1.43.2	F	361987568	144	3
CF_1.47.1	F	140182544	84	3
CF_1.47.2	F	362014448	84	3
CF_1.51.1	F	140198224	144	3
CF_1.51.2	F	362030128	144	3
CF_1.57.1	F	140238544	144	3
CF_1.57.2	F	362070448	144	3
CF_1.61.1	F	140265424	144	3
CF_1.61.2	F	362097328	144	3
CF_1.65.1	F	140292304	144	3
CF_1.65.2	F	362124208	144	3
CF_1.260.1	F	140709680	56	3
CF_1.260.2	F	362541584	56	3

Illustration 4 *mcBrowser*: Tree and full representation of directory

5. A Python API for MemCom

The *Python* API for *MemCom* represents one of the required building blocks for the new *B2000* framework. Based on the *Python* philosophy, it differs significantly from the *Fortran* and *C* library function calls. Thus, a database is represented by the *Python* object *memcom* rather than by a handle. Function calls are replaced by operators. Example: Read the dataset ADIR in an array in memory. With the *C* API this becomes

```
mcInt32 *adir;
adir=(mcInt32)mcDBgetset(handle,"ADIR","I",0);
```

handle being the *MemCom* database file handle. In *Python* the same operation is achieved with

```
adir=db['ADIR'][:]
```

db being the *memcom* object relating to the database file.

The API is implemented in the form of a *Python* module. It is imported in *Python* with

```
import pymemcom
```

A handy shell script invoking *Python* with *pymemcom* is furnished with the *pymemcom* package. It actually replaces the old *MemCom Monitor* (see the *pymemcom* documentation). The script is invoked without

```
pymcmon  
1.0, Python 2.2.1 (#1, Aug 30 2002, 14:22:36)  
[GCC 3.1.1] on linux2  
Type help, copyright, credits or license for more information.  
Readline and command completion included.
```

```
>>>
```

6. A new test environment for B2000

Why a new test environment?

The actual *B2000* test system works quite well for what it was supposed doing, i.e testing the correctness of some values in a set or a relational table of the *MemCom* database after the execution of a *B2000* program. However, to test the complete *B2000* environment functionalities are required. For example some processors like *b2grad* need more complex checking functions than the 'vector' and 'descriptor' statements of *b2test*, since the storage scheme of gradients has been modified and parametrised. Another example is the checking of the return value of programs and modules. More and more processors in *B2000* are now *C* or *C++* programs that return a value which has a signification in the *UNIX* world. A zero returned value indicates successful execution and a positive value flags an error. Another requirement of *b2test* is that it will be used to check the new *B2000 Python* module with the same set of tests that used to check the actual *B2000* processors. Additional improvements are also required, such as the possibility to execute only on test a the time or to have a more expressive mechanism to select tests than the actual, rather esoteric code.

To implement these new requirements, the actual *b2test* program could be extended but this is relatively complex (3500 lines). In addition, *b2test* is programmed in *Fortran77*, which is not a well adapted language for this kind of programs. Thus, we have chosen to re-implement the *b2test* in *Python*. The final prototype implements all the functionality of the old *b2test* system plus the requirements described above. It is easily maintainable and expandable (500 lines of *Python* code). Moreover the prototype makes use the standard *Python Unit Test* framework as well as *pymemcom*. Thus, existing tools based on this framework like the GUI-based test runner can be used with the new *b2test* system.

The new *b2test* program requires converting of the data of the existing tests to the new format. This was been performed automatically by means of a *Python* script which also converts the *FrameMaker* documentation to the *docbook* documentation system.

How to write tests in the new *b2test* environment

In the new format the test instructions are written in *Python*. A test is a class which inherits the base class *b2test.TestCase* and which contains a method named *runTest* or *runMultiTest*. The *runMultiTest* method is an extension to the *Python* test framework *unittest* to simulate the actual test. The *runMultiTest* method takes as parameters a list of arguments, each of them having a default value composed of a list of objects that can be a number or any other *Python* object. For each combination of the values of the default value list the test loader of *b2test* builds a test case. The test case is accepted and is put in a list of test cases only if a check of a value in the *B2000* database is performed. An example test formulation is listed in the listing below.

Recent developments in and around B2000

We show in the following the test case `stat/case1` converted to the new format. The new `b2test` environment comes with a command line and a GUI test runner. The command line execution is launched with

```
./testrunner b2test.all_tests
```

and produces output like

```
Clamped T-bar (dof6fac = 0, angle = 0, eltype = 1, processor = 1) ... ok
Clamped T-bar (dof6fac = 0, angle = 0, eltype = 1, processor = 3) ... ok
Clamped T-bar (dof6fac = 0, angle = 0, eltype = 1, processor = 2) ... ok
Clamped T-bar (dof6fac = 1e-06, angle = 0, eltype = 1, processor = 1) ...
ok
etc...
```

```
ran 23 tests in 637.065s
```

The GUI test runner is launched with

```
./gtktestrunner b2test.all_tests
```

All instructions concerning the location of the tests are listed in the `b2test.all_tests` module. An example of the GUI is displayed in the figure below. It also illustrates how errors are viewed in the GUI

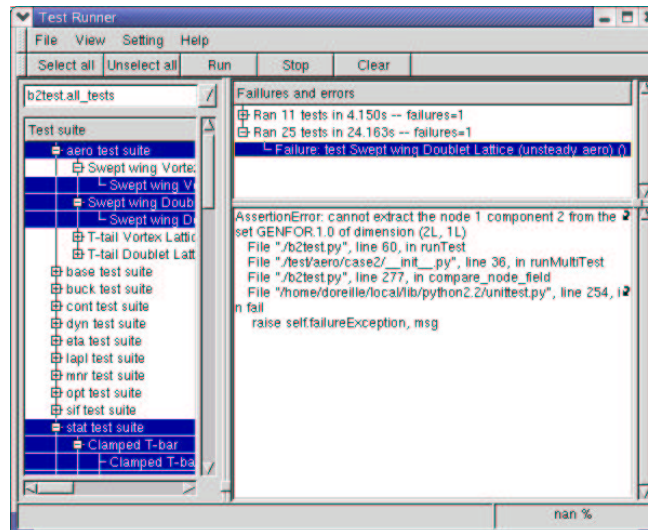


Illustration 5b2test GUI example