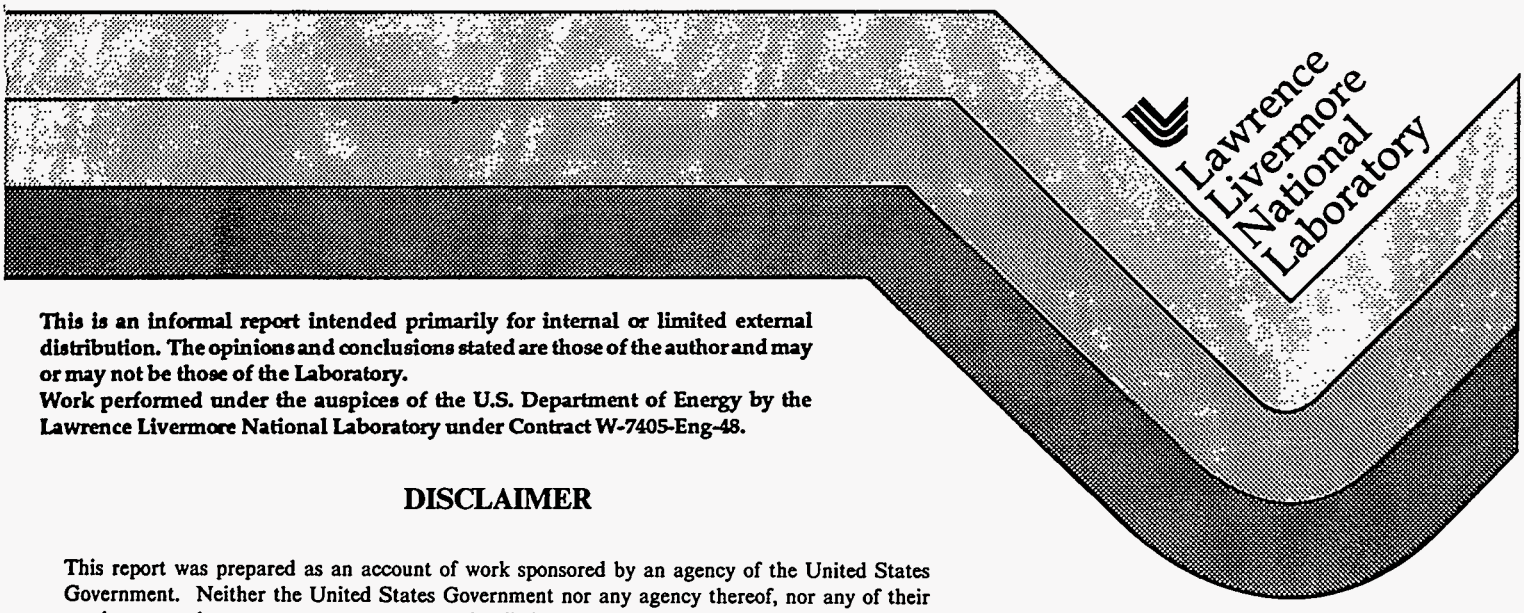
UCRL-ID-118982

Using and Programming the SUPERCODE

Scott W. Haney

June 8, 1994



DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

Using and Programming the SuperCode

Scott W. Haney

June 8, 1994

Contents

Chapter 1

Introduction

1.1 Installing the Code

If you have an account on the M-Division server *moonshine* it is quite easy to install the SUPERCODE sources in your directory space. Only a few additional steps are required to get things working on another Sun workstation.

There are two ways to install the sources on *moonshine*: (1) Install *all* of the sources, including the sources for the *SCL* and *Shell* libraries, and for the various utilities; or (2) Install only the *SC* directory, which contains the physics and engineering modules. Most users working on *moonshine* should use the latter setup since it cuts down on the amount of time required to build the SUPERCODE from scratch, and it wastes less disk space.

The following subsections give step-by-step instructions for building the SUPERCODE. In the examples, the `typewriter` typeface will be used to indicate text appearing on the computer screen. Some of the examples will show the complete line, including the prompt:

```
moonshine[1]: cd $HOME
```

The prompt, "`moonshine[1]:`," symbolizes the prompt which appears at your terminal. Yours will most likely be different. Note that these example prompts contain a number, unique to each line in the example, which may be referred to in the text.

If there are any problems, please call Scott Haney at (510) 423-6305 or Jim Crotinger at (510) 422-0259.

1.1.1 Installing on Moonshine—the Entire Code

To install the entire code on *moonshine*, follow these easy steps:

1. Go to the directory where you want to install the SUPERCODE. For example:

   ```
   moonshine[1]: cd $HOME
   ```

2. Check out the SUPERCODE source tree:

   ```
   moonshine[2]: cvs -d ~haney/CVS co SuperCode
   ```

   You should end up with a directory named *SuperCode* that is full of files.

3. Add the following lines to your `$HOME/.cshrc` file:

   ```
   setenv CPU SUN4
   setenv CVSROOT /home/moonshine/haney/CVS
   ```

   You should also add `./SUN4` to your path. Look for the "`set path`" line in the .cshrc file, and add `./SUN4` somewhere near the beginning. For example:

   ```
   set path=(. ./SUN4 ~/bin <rest-of-path>)
   ```

   An example `.cshrc` file is contained in `Docs/scode.cshrc`. After adding these lines, type

   ```
   moonshine[3]: source $HOME/.cshrc
   ```

4. Build the SUPERCODE:

   ```
   moonshine[4]: cd SuperCode
   moonshine[5]: MakeImake
   moonshine[6]: MakeTopMakefile
   moonshine[7]: make World
   ```

The *make* will take roughly 20 minutes, depending on the load on the machine.

5. You are now the proud owner of a new SUPERCODE! To run it, type

```
moonshine[8]: cd SC
moonshine[9]: SuperCode
```

1.1.2 Installing on Moonshine—SC Only

Install only the SC portion of the code on *moonshine* as follows:

1. Go to the directory where you want to install the *SC* directory. For example:

```
moonshine[1]: cd $HOME
```

2. Check out the *SC* directory:

```
moonshine[2]: cvs -d ~haney/CVS co SC
```

You should end up with a directory named *SC* that is full of files.

3. Add the following lines to your $HOME/.cshrc file:

```
setenv CPU SUN4
setenv CVSROOT /home/moonshine/haney/CVS
```

You should also add ./SUN4 to your path. Look for the "set path" line in the .cshrc file, and add ./SUN4 somewhere near the beginning. For example:

```
set path=(. ./SUN4 ~/bin <rest-of-path>)
```

After adding these lines, type

```
moonshine[3]: source $HOME/.cshrc
```

4. Build the SUPERCODE:

```
moonshine[4]: cd SC
moonshine[5]: /usr/local/SuperCode/bin/MakeTopMakefile
moonshine[6]: make Makefiles
moonshine[7]: make
```

   The *make* will take roughly 20 minutes, depending on the load on the machine.

5. You are now the proud owner of a new SUPERCODE! To run it, type

```
moonshine[8]: SuperCode
```

6. If you want the *Docs* directory, you can also check it out:

```
moonshine[9]: cvs -d ~haney/CVS co Docs
```

1.1.3 Installing on another Sun SPARCstation

Install the code on another Sun SPARCstation as follows:

1. Log on to *moonshine*.

2. Go to a directory where you would like to check out a copy of the SUPERCODE. For example:

```
moonshine[1]: cd $HOME
```

3. Check out the SUPERCODE source tree:

```
moonshine[2]: cvs -d ~haney/CVS co SuperCode
```

   You should end up with a directory named *SuperCode* that is full of files.

4. Tar up the *SuperCode* for network transport:

```
moonshine[2]: tar cf sc.tar SuperCode
```

5. Log on to your SPARCstation and ftp *sc.tar*.

```
mymachine[1]: ftp moonshine.llnl.gov
ftp> bin
ftp> get sc.tar
ftp> ^D
```

6. Add the following line to your `$HOME/.cshrc` file:

```
setenv CPU SUN4
```

   You should also add `./SUN4` to your path. Look for the "`set path`" line in the .cshrc file, and add `./SUN4` somewhere near the beginning. For example:

```
set path=(. ./SUN4 ~/bin <rest-of-path>)
```

   An example `.cshrc` file is contained in `Docs/scode.cshrc`. After adding these lines, type

```
mymachine[2]: source $HOME/.cshrc
```

7. Build the *imake* utility:

```
mymachine[3]: cd SuperCode
mymachine[4]: MakeImake
```

8. If you are still using Sun C++ 2.0, you must edit the imake configuration file for the Sun:

```
mymachine[5]: emacs Utilities/Imake/config/sun.cf
```

   Look for the line defining `CPlusPlusMinorVersion` and change the definition from 1 to 0. Also, the path required to find C++ may be different than that in *cshrc.scode*.

9. Build the SUPERCODE:

```
mymachine[6]: MakeTopMakefile
mymachine[7]: make World
```

The last command *make* will take roughly 20 minutes, depending on the load on the machine.

10. You are now the proud owner of a new SUPERCODE! To run it, type

```
mymachine[8]: cd SC
mymachine[9]: SuperCode
```

1.2 Directory Organization

Users who install the entire SUPERCODE distribution will be presented with over 200 source files and 25 sub-directories. They are laid out as follows:

- SuperCode:
  - Docs:
  - RLStream:
    - SUN4:
  - SC:
    - SUN4:
  - SCL:
    - SUN4:
  - Shell:
    - SUN4:
  - Utilities:
    - Imake:
      - config:
        - SuperCode:
      - src:
        - SUN4:
    - MGen:
      - SUN4:
    - Mppl:
      - SUN4:
    - Readline:
      - SUN4:

The *SuperCode/Docs* directory contains the LaTeX files used to generate this manual. The manual can be produced by typing "`latex SCManual`". The *SuperCode/SC* directory is of most interest to module contributors. This is where the source code for modules goes and where the SUPERCODE can be executed. There are also several example shell language programs here. The *SuperCode/SCL* directory contains a C++ class library used to build the SUPERCODE. The *SuperCode/RLStream* and *SuperCode/Utilities/Readline* directories contain files used to implement the SUPERCODE command line editing features. The *SuperCode/Shell* directory contains the C++ source code for the shell. The *SuperCode/Utilities/Imake* directory contains files used by the *imake* utility. The *SuperCode/Utilities/MGen* directory contains the source for the SUPERCODE module description file reader. Finally, the *SuperCode/Utilities/Mppl* directory contains the source for an enhanced version of MPPL used in module development. The various *SUN4* directories hold object files and the like. Hopefully, this brief discussion will help with navigation through the directory structure.

Chapter 2

The Shell

2.1 Executing the Code

Until the graphical front-end is developed, the SUPERCODE is run in a command-line mode. This version of the SUPERCODE can be most simply executed by typing

```
SuperCode
```

After a bit, the SUPERCODE shell responds with its prompt

```
In[1]:
```

In this manual, we will always use **bold face** to represent text the computer types and `typewriter face` to represent text the user types. The number in the prompt refers to line number of interactive input currently being entered. The SUPERCODE is an interactive code so a run consists of alternating user inputs and code results. For example.

```
In[1]:
1 + 1;
  2
In[2]:
2 * (3 + 4);
  14
In[3]:
quit
```

The quit directive is used to terminate the session.

It is possible to read statements into the SUPERCODE via files. For example, if the file *test.sc*[1] contained the lines

```
1 + 1;
2 * (3 + 4);
```

and the SUPERCODE was executed with the command

```
SuperCode test.sc
```

the SUPERCODE would respond as if the user had typed the lines in *test.sc* directly at the terminal:

```
  2
  14
In[1]:
```

It is also possible to read files into the SUPERCODE at any time during a session using the include directive

```
In[1]:
include "test.sc"
  2
  14
In[2]:
```

Notice that double quotation marks around the file name are required here. For compatibility with the C language convention, #include can be used in place of include.

Files can contain include directives. In this case, processing of the current file is temporarily suspended while the include'd file is read in. A file can also contain a quit directive. By placing a quit directive in a file, the SUPERCODE can be executed in a batch mode similar to conventional codes.

When the SUPERCODE starts up, it looks for a file 'SuperCode.sc'. If it can find it, the SUPERCODE reads the statements in that file before any user-specified files. This start-up file is useful for defining variables and routines are should exist for every run to the SUPERCODE.

In any interactive code, errors are bound to occur. The SUPERCODE shell is capable of detecting exceptions and returning to a state ready to accept new input. For example

[1] By convention, SUPERCODE shell statement files possess the suffix .sc

```
In[1]:
1 + * 1;
### Error 12: Misplaced token.
        I wasn't expecting to see the operator '*' here.
        File "Interactive"; Line 1 # caused the exception.
        File "Expression.cc"; Line 1072 # generated the exception.
In[2]:
```

The first few lines describe the type of exception. Then, the file and line number of the shell language instruction that actually caused the exception is given[2]. Finally, the file and line number of the instruction in the SUPERCODE source that generated the exception is given. This last piece of information is of most use to SUPERCODE implementors.

Three types of exceptions can be generated. These are, in order of increasing severity:

- *Warnings.* Indicating a potential problem that is not serious enough to stop execution of the current set of shell statements.
- *Errors.* Indicating a problem that requires the execution of the current set of shell statements to cease.
- *Fatals.* Indicating a problem that requires the execution of the SUPERCODE to cease.

Fatal errors are generally indicative of a programming error. Therefore, please note the error message along with what you were typing when the fatal error occurred and contact Scott Haney at (510)-423-6308.

2.2 Command Line Editing and History

The SUPERCODE shell allows the user to use the cursor keys and control keys to edit the line which he is typing and to recall lines which have been previously entered. Here is a summary of the features

- Use the up and down arrows (or the EMACS control sequences, Control-P and Control-N) to scroll back and forth through lines previously entered.

[2]File "Interactive" refers to interactive input

- Left and right arrows (or Control-B and Control-F) can be used to move within the text of a given command.
- Control-A moves to the beginning of a command, Control-E moves to the end.
- Control-R starts an EMACS-style search back through the previously entered commands. Control-S searches forward (which is only useful if you've already moved back in the buffer).
- Try out other EMACS editing keys such as Control-K, Control-Y, Control-D, Esc-D, etc. Most work.
- Hit the TAB key after typing the beginning of an include file name and the SuperCode will try to complete it for you. For example, typing "`include "cd`" followed by a TAB will cause the SuperCode to finish the string, resulting in "`include "cda.sc"`" appearing at the prompt.

The command line editing and history features were added to the SUPERCODE shell via the GNU Readline Library from the Free Software Foundation (FSF). Full documentation on these features are described in detail in the FSF's *GNU Readline Library Manual.* If you are interested in obtaining these, please contact Scott Haney or Jim Crotinger.

2.3 The Shell Language

Conventional codes typically read input files consisting of a list of data values. The SUPERCODE, on the other hand, reads input consisting of a list of executable statements. These statements are high-level constructs that allow

- Declaration of variables of various types.
- Arithmetic involving variables and constants.
- Tests and looping.
- Subroutines and functions.
- Access to compiled computational routines and modules.

The SUPERCODE shell language is based on the C++ programming language [1], which was developed by Bjarne Stroustrup in 1985. C++, in turn, is an extension of the popular C programming language [2], developed by Dennis Richie and Brian Kernigan in the early 1980s.

The balance of this section will briefly describe the main features of the SUPERCODE shell language. Despite the fact that the shell language differs significantly from FORTRAN, this section, along with a little bit of actual practice with the SUPERCODE should allow physicists and engineers to quickly pick up the syntax. If further information is required, please see one of the numerous books on C [3] or C++ [4, 5, 6]. In many cases, C or C++ examples from these books can executed directly in the shell.

2.3.1 Input Format

Unlike FORTRAN, which is sensitive to statement spacing, the shell language is completely free-format. Accordingly, statements can be spread over several lines and spaces, tabs, etc are generally unimportant. For example,

```
In[1]:
1 +
1
;
  2
In[4]:
```

is ugly but completely legal. This flexibility comes at a small cost though: a semi-colon is usually required to signal the end of a source line. In the above example, notice how the result of the calculation and the new prompt are not displayed until `1 + 1;` is completely entered. The shell will patiently wait until either a correct shell language statement or a syntax error is entered. If you have typed something and you think the shell should be doing something, it is probably waiting for the semi-colon you forgot to type.

2.3.2 Comments

To aid in documentation, the shell recognizes comments. For example the sample file *test.sc* discussed above can be modified to

```
/*
 * test.sc:  a very simple shell program
 */

1 + 1; // A very simple calculation.
2 * (3 + 4);
```

There are two types of comments allowed by the shell and both are present in the modified file. The first type delimits the text of the comment with the tokens /* and */. Everything between these tokens is counted as a comment and ignored by the shell. The second type, signaled by the //, indicates that the text following this token to the end of the line is a comment. Comments can also be typed during interactive input

```
In[1]:
/*
 * test.sc:  a very simple shell program
 */

1 + 1; // A very simple calculation.
  2
In[6]:
2 * (3 + 4);
  14
In[7]:
```

Note that the shell will not return a new prompt until an executable statement is given.

2.3.3 Declarations

FORTRAN's implicit typing rules generally make it unnecessary to explicitly specify the types of variables. This has the advantage of reducing typing but has the disadvantage of being error-prone. For example, forgetting that a variable named mu0 is, by default, an integer, causes big problems when that variable is meant to represent the permittivity of free space ($4\pi \times 10^{-7}$). To combat this sort of bug, the shell language requires that each and every variable be declared to have a particular type.

Basic Types

There are two fundamental types of variables: `Real` and `Integer`. The simplest types of declarations are written

```
Real a;
Integer i;
```

These two declarations create the real variable `a` and the integer variable `i`. These variables can be assigned values and manipulated in arithmetic calculations:

```
In[1]:
a = 2.3;
In[2]:
a * (2 + 1.2);
  7.36
In[3]:
i = a + 27;
In[4]:
i;
  29
```

Shell variable names must start with either a letter or an underscore. After that, any combinations of letters, numbers, and underscores can follow. The maximum variable name length is currently 256 characters. Unlike FORTRAN, the shell *is* sensitive to case so `a` is a different variable than `A`.

More elaborate declarations are possible. For instance, one can declare several variables at once by listing the variable names separated by commas

```
Real x, y, z;
Integer l, m;
```

Sometimes, it is also desirable to initialize variables directly in the declaration

```
Real a = 2.3 + (2 * 1.2);
Integer i = a + 27;
```

Finally, one also occasionally wishes to declare a variable that should not be modified. This can be done by prepending the `const` keyword to the beginning of the declaration

```
const Real pi = 3.14159;
```

Variables declared in this manner cannot appear on the left hand side of an equals sign. If you try to assign to a const variable, an error is generated. As a result, const variables *must* be initialized when they are declared (if you forget, the shell will generate an error).

The shell also supports a fairly unique type of variable called a *reference*. Reference variables are declared by placing an ampersand in front of the variable name. Therefore, the declaration

```
Real &aref = a, b;
```

declares a reference variable aref and an normal real variable b. A reference is an alias for another variable. In the example above, aref is a reference to the variable a. This means that all operations done with and to aref are equivalent to performing those operations with or to a (and visa versa):

```
In[1]:
Real a = 4.12, &aref = a;
In[2]:
aref;
  4.12
In[3]:
aref = 9.58;
In[4]:
a;
  9.58
In[4]:
a = 2.01;
In[5]:
aref;
  2.01
```

A non-const reference cannot be an alias for a constant. However, it is possible declare a const reference that is. It is also illegal to have a non-const reference variable alias a variable of a different type. Here are some examples:

```
Integer i;
Real a;
Integer &j = i; // Legal
Integer &k = i; // Illegal
```

```
const Integer &k = 1; // Legal
Integer &l = a; // Illegal
const Integer &l = a; // Legal
```

You may be wondering what reference variables are actually good for. A later section will describe how they are extremely useful as function or subroutine parameters.

For convenience, the shell automatically declares a number of work variables including the Integers _i, _j, _k, _l, _m, _n and the Reals _a, _b, _c, _d, _e, _f.

Other Types

In addition to real numbers and integers, the shell currently supports a variety of other types including

- `Complex`. Complex numbers.
- `ComplexArray3`. Three-dimensional arrays of complex numbers.
- `ComplexMatrix`. Two-dimensional arrays of complex numbers.
- `ComplexVector`. One-dimensional arrays of complex numbers.
- `Fstream`. Stream input or output from files.
- `IntegerArray3`. Three-dimensional arrays of integers.
- `IntegerMatrix`. Two-dimensional arrays of integers.
- `IntegerVector`. One-dimensional arrays of integers.
- `Ostream`. Stream output to the terminal.
- `RealArray3`. Three-dimensional arrays of real numbers.
- `RealMatrix`. Two-dimensional arrays of real numbers.
- `RealVector`. One-dimensional arrays of real numbers.
- `String`. Character strings.

- **Subroutine.** Aliases for subroutines taking no arguments.
- **Void.** No type.

These types will be described in more detail later in this chapter.

Functions

In addition to variables, the shell allows the declaration of sophisticated functions and subroutines. Function declarations are similar to variable declarations except

- Only one function can be declared at once.
- The types of arguments the function expects to get must be specified.
- The statements making up the function must be supplied.

For example, the declaration

```
In[1]:
Real foo(Integer j, Integer k)
{
  :
}
```

says that `foo` has the type "function expecting two `Integer` arguments and returning a `Real`". The two variables `j` and `k` are known as *formal arguments.* The unspecified stuff inside the curly brackets is the statements the function is supposed to execute when it is called. The return type can be any of the possibilities we just discussed (including a reference). Similarly, there can be any number of arguments of any type. Note that even if there are no arguments, a pair of parentheses must follow the function name.

To declare a subroutine have it return type `Void`:

```
In[1]:
Void bar(const Real &xyzzy)
{
  :
}
```

This is the only place `Void` can appear.

A number of useful arithmetic functions have been pre-defined including `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, and `atan`.

Subroutine Aliases

On occasion, it is very convenient to have a variable that serves as an alias of a function. Then, this variable can be passed to another function and used to call the original function. The shell currently has a limited capability to do this using the `Subroutine` type. The `Subroutine` type is an alias to functions of the form

```
Void vFun()
{
  :
}
```

and a `Subroutine` variable can be instantiated with a declaration of the form

```
Subroutine aAa = vFun;
```

After this declaration, `aAa` can be used interchangedly with `vFun`. Unlike references, which have similar sorts of properties, it is possible to assign a new function to alias after creation

```
In[1]:
Void vFun() { }
In[2]:
Void vFun1() { }
In[2]:
Subroutine aa, bb = vFun1;
In[3]:
aa = bb; bb = vFun;
In[4]:
```

We will see all sorts of uses for `Subroutine` variables in later chapters.

2.3.4 Expressions

The shell understands a wide variety of arithmetic operators. These operators are summarized in Table 2.3.4. Note that all operators between a set of horizontal lines in the table have the same precedence. They also have the same grouping (or associativity) properties. As in FORTRAN, the order of evaluation can always be modified by grouping things in parentheses. The balance of this section will describe the operators in more detail.

Function Call

Function calls have the syntax

expression1(*expression-list*)

where *expression1* is an expression evaluating to "function returning T", T is some type (including `Void`), and *expression-list* is a comma separated list of expressions which constitute the actual arguments of the function. Note that *expression-list* may be empty but the parentheses are still required. Finally, there is no need for a `call` keyword even for `Void` functions (i.e., subroutines).

The simplest *expression1* is the name of a previously defined function

```
In[1]:
Integer i = 1;
In[2]:
Void foo(Integer j, Integer k)
{
  i = j + k;
}
In[6]:
foo(3, 5);
In[7]:
i;
  8
In[8]:
```

although more complicated situations can arise. For instance, suppose a is a function taking no arguments and returning a `Subroutine` that is an alias for a `Void` function b. Then, the surprising statement

```
a()();
```

Table 2.1: Shell Operator Summary

Operator(s)	Grouping
Function call () Postincrement ++ Postdecrement --	left to right
No operation + Negation - Logical negation ! One's complement ~ Preincrement ++ Predecrement --	right to left
Multiplication * Division / Remainder %	left to right
Addition + Subtraction -	left to right
Left bit shift << Right bit shift >>	left to right
Relational <, >, <=, >=	left to right
Equal to == Not equal to !=	left to right
Bitwise AND &	left to right
Bitwise Exclusive OR ^	left to right
Bitwise Inclusive OR |	left to right
Logical AND &&	left to right
Logical OR ||	left to right
Conditional ? :	right to left
Assignment =, *=, /=, %= +=, -=, <<=, >>= &=, ^=, |=	left to right
Comma ,	left to right

actually ends up calling b!

When a function is called, each formal argument of a function is initialized with the actual argument at the same location in *expression-list*. Inside the function, it is possible to change the values of any non-const formal argument but these changes will not affect the values of the actual arguments unless the arguments are references. This is different from FORTRAN where all changes to the formal arguments change the actual arguments. Here is an example:

```
In[1]:
Integer i = 1;
In[2]:
Void foo(Integer k) { k = 3; }
In[3]:
Void bar(Integer &k) { k = 3; }
In[4]:
foo(i);
In[5]:
i;
  1
In[6]:
bar(i);
In[7]:
i;
  3
In[8]:
```

This, incidentally, is the main use for references: they allow functions to modify the actual arguments passed to them.

As we mentioned in the last section, it is possible to declare two functions with the same name but taking different arguments. How does the shell decide which function to call? This decision is made by comparing the actual arguments with the formal arguments of all of the functions with the same name. The actual procedure is fairly complicated but it can be summarized as follows:

1. Eliminate all possibilities that expect the wrong number of arguments.

2. For each argument, find the set of functions whose formal argument best matches the actual argument. An exact match (e.g. a `Real` with a `Real`) is better that a conversion (e.g. a `Real` with an `Integer`). If a

function has formal arguments that cannot be initialized by the actual arguments (e.g., a `RealMatrix` formal argument cannot be initialized by an `Integer`) that function is eliminated from the competition.

3. Form the intersection of the sets of functions from each of the arguments that yield an equally good match. If the resulting set consists of one and only one function, this is the one to call. If the resulting set is empty, this is an error. If the resulting set has more than one function in it, the call is ambiguous and an error is generated.

This isn't as hard as it sounds. Consider the following

```
In[1]:
Integer i;
In[2]:
Void f(Integer l, Integer m) { i = 1; }
In[3]:
Void f(Real l, Real m) { i = 2; }
In[4]:
Void f(Integer l, Real m) { i = 3; }
In[5]:
f(1, 1);
In[6]:
i;
  1
In[7]:
f(1.0, 1.0);
In[8]:
i;
  2
In[9]:
f(1, 1.0);
In[10]:
i;
  3
In[11]:
f(1.0, 1);
  ### Error 40: Ambiguous call to overloaded function.
       Multiple versions of 'f' can accept the values passed.
       File "Interactive"; Line 12 # caused the exception.
```

File "TypeCheck.cc"; Line 957 # generated the exception.
In[12]:

The last call is ambiguous because `f(Real, Real)` has an exact match on the first argument and it requires an `Integer` ⟶ `Real` conversion on the second while `f(Integer, Integer)` has an exact match on the second argument and it requires a `Real` ⟶ `Integer` conversion on the first.

Postincrement and Postdecrement

These operators return the value of the operand and then add (++) or subtract (--) 1 from it. The operand must be what is called an *lvalue*. An lvalue is basically something that can appear on the left hand side of an equals sign. For example, a non-`const` variable is an lvalue while the number 3 is not. The result of a postincrement or decrement is not an lvalue.

Unary Plus

A unary plus in front of an operand just returns that operand. The result is not an lvalue.

Unary Minus

A unary minus in front of an operand returns the negation of that operand. The result is not an lvalue.

Logical Negation

The result of the unary logical negation operator (!) is of type `Integer`. If the operand is 0, the operator returns 1, if the operand is non-zero, the operator returns 0. The result is not an lvalue.

One's Complement

The operand of the one's complement operator (∼) must be an `Integer`. The result is an `Integer` with all of the 0 bits in the binary representation of the operand switched to 1 and visa versa. The result is not an lvalue.

Preincrement and Predecrement

These operators add (++) or subtract (--) 1 from the operand and return than new value. The operand must be an lvalue and the result is not an lvalue.

Multiplication, Division, and Remainder

These are all binary operators that return non-lvalues. Multiplication (*) and division (/) work in the way you'd expect. The remainder operator (%) only works with operands of type `Integer`. The result is what is left over after dividing the first operand by the second. Therefore, `(a/b) * b + a % b` equals a.

Addition and Subtraction

These are binary operators that return non-lvalues. Addition (+) and subtraction (-) work as they always have.

Left and Right Bit Shifts

These are binary operators that expect `Integer` operands and return non-lvalues. These operators shift the bits of the left operand the number of places specified by the right operand to the left (<<) or right (>>). Both shifts fill in with zeros as the number is shifted. For example `1 << 3` equals 8 and `16 >> 2` equals 4.

Relational Operators

These are binary operators that return non-lvalues of type `Integer`. The possible comparisons are less-than (<), greater-than (>), less-than-or-equal (<=), or greater-than-or-equal (>=) They compare the two operands and return 1 if the condition evaluates to true and 0 if the condition evaluates to false. You can count on these values so relational comparisons can be used in arithmetic operations.

Equality Operators

These are binary operators that return non-lvalues of type `Integer`. The possible comparisons are equal-to (==) or not-equal-to (!=). They compare

the two operands and return 1 if the condition evaluates to true and 0 if the condition evaluates to false. You can count on these values so equality comparisons can be used in arithmetic operations.

Bitwise Operators

These are binary operators that expect `Integer` operands and return non-lvalues. These operators perform the bitwise OR (|), bitwise exclusive-OR (^), or bitwise AND (&) boolean functions on the operands.

Logical AND

This binary operator (&&) returns a non-lvalue of type `Integer`. The result is 1 if both operands are non-zero and 0 if one of the operands is 0. If the first operand is zero, the second is *not* evaluated.

Logical OR

This binary operator (||) returns a non-lvalue of type `Integer`. The result is 1 if either operand is non-zero and 0 if both of the operands are 0. If the first operand is non-zero, the second is *not* evaluated.

Conditional Operator

This is the world's first ternary operator. It has the syntax

expression1 ? *expression2* : *expression3*

This operator works by first evaluating *expression1*. If this is non-zero the result is whatever *expression2* evaluates to. Otherwise, it is is whatever *expression3* evaluates to. For example, consider the expression

```
a = (b > 3) ?  12.5 :  1.25;
```

If `b == 4`, `a` is assigned 12.5, if `b == 1`, `a` is assigned 1.25.

Assignment Operators

The shell supports a bunch of assignment operators that all require an lvalue as the left operand and return an lvalue with the type of the left operand. The possible assignments are simple-assignment (=), multiply-and-assign (*=), divide-and-assign (/=), modulo-and-assign (%=), add-and-assign (+=), subtract-and-assign (-=), shift left-and-assign (<<=), shift right-and-assign (>>=), AND-and-assign (&=), inclusive OR-and-assign (|=), or exclusive OR-and-assign (^=). In simple assignments, the value of the right operand (after possible conversions) replaces that of the left operand. The behavior of the other assignments of the form

expression1 op= expression2;

is equivalent to

expression1 = *expression1 op expression2*;

except that *expression1* is only evaluated once.

Comma Operator

This binary operator (,) first evaluates the first operand and discards the result. Then, it evaluates the second operand and that is the result. The result is an lvalue if the second operand is an lvalue The effect of the first operand is felt via side effects. For instance, the expression

```
a = (b = 11, b + 7);
```

ends up setting a to 18 and b to 11. Be sure to apply parentheses liberally with the comma operator since it is the lowest precedence operator around. Also, take care when using it in function calls since the shell assumes the commas there separate the arguments.

2.3.5 Statements

The SUPERCODE shell interprets statements until the `quit` directive is issued. The statements that the shell understands are summarized in Table 2.3.5. The balance of this section will describe these statements in more detail.

Table 2.2: Shell Statement Summary

Category	Syntax
Declaration	*Discussed in previous section*
Null	;
Expression	*expression* ;
Selection	`if` (*expression*) *statement1* `if` (*expression*) *statement1* `else` *statement2*
Looping	`while` (*expression*) *statement* `for` (*expression1* ; *expression2* ; *expression3*) *statement*
Compound	{ *statements* }
Jump	`break` ; `continue` ; `return` ; `return` *expression* ;

Null Statement

A null statement consists of a single semi-colon. It is the simplest statement known by the shell. Null statements don't do anything but they are useful occasionally.

Expression Statement

Most of the statements you type into the shell will be expression statements. Expression statements consist of an expression (usually an assignment or a function call) followed by a semi-colon. All side effects of an expression statement are completed before the next statement is executed. We have already given numerous examples of this sort of statement so none will be supplied here.

Selection Statements

Selection statements choose a particular flow of control. Based on Table 2.3.5, we see that the simplest selection statement can look something like

```
if (a > 3 && a < 6)
  b = 47;
```

If the condition inside the parentheses is non-zero, the following statement (`b = 47;`) is executed, otherwise control is transferred to the statement following that.

It is also possible to add a statement to execute if the condition is zero. This is accomplished with the `else` keyword:

```
if (a > 3 && a < 6)
  b = 47;
else
  b = -12;
```

In this case, if `a == 4`, the variable `b` is assigned 47 whereas if `a == 1`, `b` is assigned -12.

The introduction of the `else` keyword creates an ambiguity when reading nested `if` statements. Consider

```
if (x <= 7 || x > 22)
  if (x == 33)
    y = -x * z;
else
  y = 3.5 * z + 4;
```

Despite the suggestive indentation, the `else` clause does not belong with the initial `if` statement; instead, it goes with the second. The rule is that an `else` always goes with the last `else`-less `if`. We will soon see how to achieve the flow of control indicated by the indentation.

There is a slight subtlety to remember when typing in `if` statements interactively. After shell sees the semi-colon after the expression `b = 47` it does not know whether or not an `else` clause will follow or not. Therefore, the shell will continue to wait for input in order to decide. If you don't have an `else` following the `if` statement, simply type a semi-colon (i.e., a null statement) and the shell will evaluate the `if`:

```
In[1]:
Integer i = 3, j;
In[2]:
if (i == 3)
```

```
  j = 4;;
In[4]:
j;
  4
```

The second semi-colon after the expression `j = 4` is the null statement.

Looping Statements

As Table 2.3.5 suggests, the shell understands two looping constructs: `while` statements and `for` statements.

An example of a `while` statement is

```
Integer i = 0, j = 0;
while (i < 5)
  j += i++;
```

In a `while` loop, the statement is executed repeatedly until the expression inside the parentheses after the `while` keyword evaluates to zero. This test takes place before the execution of the statement. Using these rules, it is fairly easy to determine that `j == 10` after the loop is executed.

The `for` loop is a more compact way of implementing the same algorithm as a `while` loop. An `for` loop equivalent to the `while` loop above is

```
Integer i, j;
for (i = 0, j = 0; i < 5; i++)
  j += i;
```

As you can see, there are three expressions inside the parentheses after the `for` keyword. The first expression specifies initialization for the loop (note the use of the ubiquitous comma operator). The second expression specifies a test made before each iteration. If this expression evaluates to zero, the loop is exited. The third expression is evaluated after each iteration. It usually serves to increment a counter.

Compound Statements

It would be very inconvenient if loops could consist of only one statement. Happily, the shell supports a construct called a *compound statement* that solves

this problem. Simply put, a compound statement is a series of statements surrounded by curly brackets. Compound statements are very powerful because they can be used anywhere a statement can be used. Therefore, the `while` loop example above can be written in a slightly more readable form

```
Integer i = 0, j = 0;
while (i < 5)
  {
    j += i;
    i++;
  }
```

The bodies of functions are actually compound statements. Therefore, we can define a function having more than one line. For instance

```
Void swap(Real &x, Real &y)
{
  Real z;
  z = x;
  x = y;
  y = z;
}
```

is a subroutine that exchanges the value of two `Real` numbers.

Notice the presence of a declaration in the compound statement making up the body of swap. One can place any number of variable declarations anywhere inside a compound statement. However, function declarations cannot be placed inside a compound statement.

In addition to providing a way to group statements, a compound statement also introduces what is called a *scope*. Scope is used to determine the meaning of a name as it is used at a particular place in a shell program. Up to now, this has been obvious

```
In[1]:
Integer i;
In[2]:
Integer j = 2;
In[3]:
i = 3 + j;
In[4]:
```

Clearly, the expression in **In[3]** refers to the `i` declared in **In[1]** and the `j` in **In[2]**. However, suppose we have the situation

```
In[1]:
Integer i = 0;
In[2]:
Integer j = 2;
In[3]:
{
  Integer i;
  i = 3 + j;
}
In[7]:
```

What is the value of `i` after this set of statements? It turns out that the answer is `i == 0`.

The scoping rules used in the SUPERCODE shell are relatively simple

1. The scope of a name extends from the point of declaration to the end of the compound statement the declaration resides in. This is also the lifetime of any physical storage required for the variable.

2. A declaration of a name in a compound statement hides declarations of the same name from enclosing scopes.

3. After exit from the compound statement, the name resumes its previous meaning.

To understand these rules, consider the following code:

```
Integer jj = 6;
Void g(Integer kk)
{
  Integer ii = jj;
  Integer jj = 13;
  ii = jj;
}
```

The variable `ii` inside g is initialized using the global version of `jj`. Hence, its value is initially 6. When the local version of `jj` is declared, the global `jj` is hidden so when `ii` is set on the next line, it is set to 13. The storage

for the local variables ii and jj exists until the closing curly bracket of g. The same goes for the formal parameter kk which—although it doesn't look that way—has the same scope as the local variables. The global version of jj remains in existence until the SUPERCODE execution ends.

One might be tempted to wonder if these scoping rules are really necessary. Well, in big programs they are. The only alternative is to ban name conflicts. In other words, if there was a global variable named i, one could not create local variables with the same name. In a multi-programmer project there is no way to ensure that such conflicts do not routinely cause big problems. The scope rules are relatively easy to live with after some practice.

Jump Statements

Jump statements unconditionally transfer control. The shell understands three of these: break statements, continue statements, and return statements.

The break statement can occur only inside loops. It causes immediate termination of the enclosing loop. The continue statement also can occur only inside loops. It causes the next iteration of the loop to start immediately.

The return statement can only appear inside functions. A return statement without an expression can only appear inside a function returning type Void (i.e., a subroutine). It causes control to return to the statement after the function call. Hitting the terminating curly bracket of a function is equivalent to executing a return statement without an expression. Again, this should only happen in a subroutine.

A return statement with an expression can be used only in functions returning values. This value is returned to the caller. If required, the value is converted to the return type of the function. To see how return statements work consider a function for calculating factorials

```
In[1]:
Integer factorial(Integer n)
{
  return (n > 1) ?  n * factorial(n - 1) :  1;
}
In[5]:
factorial(4);
  24
In[6]:
```

```
factorial(6);
  720
In[6]:
```

Notice that shell functions can be recursive!

It is also possible to return references:

```
In[1]:
Real x = 0;
In[2]:
Real &reftest()
{
  return x;
}
In[6]:
reftest() = 3.14159;
In[7]:
x;
  3.14159
In[8]:
```

This is certainly an interesting way to assign a value to x.

2.4 Classes

The SUPERCODE shell is able to manipulate sophisticated entities called *objects*. Due to the recent interest in a methodology called object-oriented programming, a number of definitions of *object* are available in the literature. We use the one by Booch [7]: "An object has state, exhibits some well-defined behavior, and has a unique identity." A simple example of an object is a variable c holding a complex number. The variable has a state ($c = 3 + 4i$, for instance), it has behavior governed by the rules of complex arithmetic, and it has a unique identity (c is different than another complex variable d).

An important concept closely related to that of an object is a *class*. A class is a template that describes the structure and behavior of a set of objects. A common synonym for class is type. The rest of this section will deal with the description of classes that the SUPERCODE supports.

2.4.1 Complex numbers

The class description for complex numbers as currently implemented by the SUPERCODE shell is

```
class Complex {
public:
  Complex();
  Complex(Real r);
  Complex(Real r, Real i);
  Complex(const Complex &c);
  ~Complex();

  Complex &operator=(const Complex &c);
  friend Complex operator+(Complex lc, Complex rc);
  friend Complex operator-(Complex lc, Complex rc);
  Complex operator-(Complex c);
  friend Complex operator*(Complex lc, Complex rc);
  friend Complex operator/(Complex lc, Complex rc);

  friend Fstream &operator<<(Fstream &os, Complex x);
  friend Fstream &operator>>(Fstream &os, Complex &x);
  friend Ostream &operator<<(Ostream &os, Complex x);

  Complex conj();

  Real re, im;
};
```

FORTRAN programmers will probably find the above notation to be strange. This notation is borrowed from the C++ programming language. For more details, see any of the C++ references.

A class is made up of *members* and *friends*. The `Complex` class above has two data members (the real part `re` and the imaginary part `im`), seven function members (four versions of the `Complex` function, `~Complex`, `operator=`, and `conj`), and the seven friend functions (`operator+`, etc).

The first four member functions of the `Complex` class are called *constructors*. They allow one to declare complex number is various ways. For instance,

```
Complex x1; // x is initialized to 0 + 0i
```

```
Complex x2(-12.32); // x2 is initialized to -12.32 + 0i
Complex x3(2.718, -3.142); // x3 is initialized to 2.718 - 3.142i
Complex x4(x3); // x4 is initialized to the value of x3
```

Constructors are usually called automatically by the shell. In the above example the x1 declaration generates a call to the first constructor in the class description, the x2 declaration generates a call to the second, the x3 declaration generates a call to the third, and the x4 declaration generates a call to the fourth. Like Real and Integer variables, it is also possible to alternatively use an assignment syntax to generate calls to the constructors

```
Complex x5 = 23.5; // x5 is initialized to 23.5 + 0i
Complex x6 = Complex(1, 2); // x6 is initialized to 1 + 2i
```

The declaration of x5 generates a call to the constructor taking one Real argument while the declaration of x6 generates a call to the constructor taking the Complex reference argument.

The ~Complex member function is called a destructor. Destructors are never called by the user, they are automatically called by the shell whenever a class object goes out of scope. Destructors generally are in charge of discarding memory associated with the object.

The functions with names of the form operator*op* implement simple complex number arithmetic. One does not need to call these functions explicitly. Rather, when the shell sees an operation involving Complex quantities it picks out the right version to call. For example, one can execute the following expressions

```
In[1]:
Complex a(1.2, 2.3), b(-1), c(6.1, -7.8);
In[2]:
-a * (b + c);
  (2.718, -3.142)
In[3]:
```

Eventually, the shell will support a complete set of arithmetic functions involving Complex variables. However, the simple operations described in the class description are currently the only ones allowed.

Unlike operator= etc. some member functions must be called explicitly. The syntax for accessing these member functions is

object-name . *function-member-name*(*argument-list*)

For instance, the conj member function (which returns the complex conjugate) is called as follows

```
In[1]:
Complex a(1.2, 2.3);
In[2]:
a.conj();
  (1.2, -2.3)
In[3]:
```

Data members are accessed similarly:

object-name . *data-member-name*

Hence,

```
In[1]:
Complex a(1.2, 2.3);
In[2]:
a.re; a.im;
  1.2
  2.3
In[3]:
```

The remaining friend functions operator<< allow Complex quantities to use the stream I/O facility to be described in Section~2.5. Finally, note that the shell pre-defines six Complex work variables (_ca, _cb, _cc, _cd, _ce, _cf) that can be used in expressions without the need to declare them.

2.4.2 3-D Arrays

The three-dimensional array class description is

```
class TYPEArray3 {
public:
  TYPEArray3();
  TYPEArray3(Integer dim1, Integer dim2, Integer dim3);
  TYPEArray3(const TYPEArray3 &arr);
  ~TYPEArray3();

  TYPEArray3 &operator=(const TYPEArray3 &mat);
```

```
  friend TYPEArray3 operator+(const TYPEArray3 &la, const TYPEArray3 &ra);
  friend TYPEArray3 operator-(const TYPEArray3 &la, const TYPEArray3 &ra);
  friend TYPEArray3 operator*(const TYPEArray3 &la, const TYPEArray3 &ra);
  friend TYPEArray3 operator/(const TYPEArray3 &la, const TYPEArray3 &ra);

  friend Fstream &operator<<(Fstream &os, const TYPEArray3 &arr);
  friend Ostream &operator<<(Ostream &os, const TYPEArray3 &arr);

  TYPE &operator()(Integer i, Integer j, Integer k);
  TYPE &data();

  Integer dim1();
  Integer dim2();
  Integer dim3();
};
```

where TYPE = Complex, Integer, or Real gives the type of the elements contained in the Array3.

The Array3 constructors allow the user to specify the initial size of these objects:

```
IntegerArray3 A1; // A1 initially is 0 × 0 × 0
RealArray3 A2(3, 2, 2); // A2 initially is 3 × 2 × 2
RealVector A3 = A2; // A3 initially has size/elements of A2
```

Unless initialized to another object, three-dimensional arrays are initially filled with zeros. The operator() member function allows Array3 elements to accessed using a FORTRAN-like syntax

```
In[1]:
A2(3, 1, 2) = -14.5; A2(2, 2, 1) = 9;
In[2]:
```

Furthermore, like FORTRAN, Array3 are indexed from 1. Array3 types are printed by the shell using a format compatible with the *Mathematica* symbolic algebra program

```
In[1]:
A2;
  {
  {{ 0, 0 },
```

```
    { 0, 9 },
    { 0, 0 }},
  {{ 0, 0 },
    { 0, 0 },
    { -14.5, 0 }}
  }
In[2]:
```

Eventually, the shell will support a complete set of arithmetic functions involving Array3 variables. However, currently, the limited set of operations described above is all that has been implemented. Note that `operator*` and `operator/` do elementwise multiplication and division.

The `data` member function is used to pass the location of a Array3's storage to a routine that cannot deal with the composite Array3 structure (for example, a FORTRAN library function). Array3 data is stored contiguously according to FORTRAN conventions. The `dim1`, `dim2`, and `dim3` member functions return the dimensions of the Array3.

2.4.3 Matrices

The matrix class description is

```
class TYPEMatrix {
public:
  TYPEMatrix();
  TYPEMatrix(Integer nrows, Integer ncols);
  TYPEMatrix(const TYPEMatrix &mat);
  ~TYPEMatrix();

  TYPEMatrix &operator=(const TYPEMatrix &mat);
  TYPEMatrix &operator=(TYPE s);
  friend TYPEMatrix operator+(const TYPEMatrix &lm, const TYPEMatrix &rm);
  friend TYPEMatrix operator-(const TYPEMatrix &lm, const TYPEMatrix &rm);
  friend TYPEMatrix operator*(const TYPEMatrix &lm, const TYPEMatrix &rm);
  friend TYPEMatrix operator*(TYPE s, const TYPEMatrix &mat);
  friend TYPEMatrix operator*(const TYPEMatrix &mat, TYPE s);
  friend TYPEMatrix operator%(const TYPEMatrix &lm, const TYPEMatrix &rm);
  friend TYPEMatrix operator/(const TYPEMatrix &lm, const TYPEMatrix &rm);
  friend TYPEMatrix operator/(TYPE s, const TYPEMatrix &mat);
```

```
    friend TYPEMatrix operator/(const TYPEMatrix &mat, TYPE s);

    friend TYPEMatrix operator&(const TYPEMatrix &lm, const TYPEMatrix &rm);

    friend Fstream &operator<<(Fstream &os, const TYPEMatrix &mat);
    friend Ostream &operator<<(Ostream &os, const TYPEMatrix &mat);

    TYPE &operator()(Integer i, Integer j);
    TYPE &data();

    Integer rows();
    Integer cols();
};
```

where TYPE = Complex, Integer, or Real gives the type of the elements contained in the matrix.

The matrix constructors allow the user to specify the initial size of these objects:

```
IntegerMatrix A1; // A1 initially has 0 rows, 0 cols
RealMatrix A2(3, 2); // A2 initially has 3 rows, 2 cols
RealVector A3 = A2; // A3 initially has size/elements of A2
```

Unless initialized to another object, matrices are initially filled with zeros.

The operator= functions allows one to assign matrices and scalars to matrices. If a and b are matrices, the expression a = b replaces the contents of matrix a with that of b. For this operation to work, a and b must be the same size. Assigning a scalar to a matrix (as in a = 3.14) sets all of the elements in the matrix to the scalar value.

The shell supports a fairly complete set of arithmetic functions involving matrix variables. Note that operator* and operator/ do elementwise multiplication and division while operator% does matrix multiplication. For example,

```
In[1]:
RealMatrix a(2,2), b(2,2);
In[2]:
a(1,1) = 1; a(1,2) = 2; a(2,1) = 3; a(2,2) = 4;
In[3]:
b(1,1) = -1; b(1,2) = -3; b(2,1) = 3; b(2,2) = 4;
```

```
In[4]:
a * b;
  {{ -1, -6 },
   { 9, 16 }}
In[5]:
a % b;
  {{ 5, 5 },
   { 9, 7 }}
In[6]:
```

Scalar multiplication is also supported.

The function operator& is fairly unique in that it allows matrix concatenation. For example,

```
In[1]:
RealMatrix a(2,2), b(2,2);
In[2]:
a(1,1) = 1; a(1,2) = 2; a(2,1) = 3; a(2,2) = 4;
In[3]:
b(1,1) = -1; b(1,2) = -3; b(2,1) = 3; b(2,2) = 4;
In[4]:
a & b;
  {{ -1, 2, 3, 4 },
   { -1, -3, 3, 4}}
```

For this operation to work, all the matrices involved must have the same number of rows.

It is possible to build up complicated expressions involving matrices and scalars. For example:

```
In[1]:
RealMatrix a(2,1), b(2,1), c(2,2), d(2,2);
In[2]:
d = (a & b) + 3.0 * c;
```

The operator() member function allows matrix elements to accessed using a FORTRAN-like syntax

```
In[1]:
A2(3, 1) = -14.5; A2(2, 2) = 9;
In[2]:
```

Furthermore, like FORTRAN, shell matrices are indexed from 1. Matrix types are printed by the shell using a format compatible with the *Mathematica* symbolic algebra program

```
In[1]:
A2;
  {{ 0, 0 },
   { 0, 9 },
   { -14.5, 0 }}
In[2]:
```

The `data` member function is used to pass the location of a matrix's storage to a routine that cannot deal with the composite matrix structure (for example, a FORTRAN library function). Matrix data is stored contiguously by columns to be compatible with FORTRAN. The `rows` and `cols` member functions return the dimensions of the matrix.

2.4.4 Vectors

The vector class description is

```
class TYPEVector :  public TYPEMatrix {
public:
  TYPEVector();
  TYPEVector(Integer nrows);
  TYPEVector(const TYPEVector &vec);
  ~TYPEVector();

  TYPEVector &operator=(const TYPEVector &vec);
  TYPEVector &operator=(TYPE s);

  TYPE &operator()(Integer i);
  TYPE &data();

  Integer length();
};
```

where `TYPE` = `Complex`, `Integer`, or `Real` gives the type of the elements contained in the vector.

The TYPEVector class is said to *inherit* from the TYPEMatrix class. This means that any routine that expects to be passed a TYPEVector can be passed a TYPEVector. (Note: the inverse is not true: one cannot pass a TYPEMatrix to a routine expecting a TYPEVector.) This feature means that all of the binary operations involving matrices are automatically supported for vector. Moreover, it is possible to mix matrices and vectors in expressions.

The vector constructors allow the user to specify the initial size of these objects:

```
IntegerVector A1; // A1 initially has 0 rows
RealVector A2(3); // A2 initially has 3 rows
RealVector A3 = A2; // A3 initially has size/elements of A2
```

Unless initialized to another object, vectors are initially filled with zeros. The operator() member function allows vector elements to accessed using a FORTRAN-like syntax

```
In[1]:
A2(3) = -14.5; A2(2) = 9;
In[2]:
```

Furthermore, like matrices, shell vectors are indexed from 1. Vector types are printed by the shell using a format compatible with the *Mathematica* symbolic algebra program

```
In[1]:
A2;
  {{ 0 },
   { 9 },
   { -14.5 }}
In[2]:
```

The data member function is used to pass the location of a vector's storage to a routine that cannot deal with the composite vector structure (for example, a FORTRAN library function). Vector data is stored contiguously to be compatible with FORTRAN. The length member function returns the dimensions of the vector.

2.4.5 Strings

String variables represent character string data. Their current class description is

```
class String {
public:
  String();
  String(const String &str);
  ~String();

  String &operator=(const String &str);
  friend String operator+(const String &ls, const String &rs);

  friend Fstream &operator<<(Fstream &os, const String &str);
  friend Ostream &operator<<(Ostream &os, const String &str);
};
```

The string constructors allow users to declare String objects as follows

```
String a, b(''hello''), c = ''goodbye'';
```

In this example, a is initially the null string "", b is initially ''hello'' and c is ''goodbye''. Other than stream I/O, the only operation currently supported for strings is concatenation (denoted by the '+' operator). For instance,

```
a = b + '' '' + c;
```

would result in a containing "hello goodbye".

2.5 Shell Input and Output

As we have seen, the simplest way to get information regarding a variable is to use it in an expression statement

```
In[1]:
Real x = 46.89;
In[2]:
x;
  48.89
In[3]:
```

The shell tries to be somewhat selective about the type of expressions that it outputs expressions for. In particular, the shell will not output a result when the first operator encountered is an assignment operator:

```
In[1]:
Real y;
In[2]:
y = 1.2 + 3.4;
In[3]:
y;
  4.6
In[4]:
```

This conflicts slightly with the C++ convention that assignment is just another operator but cuts down on generally unwanted output.

It is also possible to get output by using the `Ostream` class. `Ostreams` currently cannot be instantiated[3] for in the SUPERCODE shell but a pre-defined variable cout has been provided. The cout variable is used according to C++-style stream I/O conventions [4, 5, 6]. For example, to output two numbers to the terminal one simply types

```
In[1]:
Real a = 1.4, b = -1.3;
In[2]:
cout << a << '' '' << b << ''\n'';
  1.4 -1.3
In[3]:
```

As this example demonstrates, it is possible to output various types of variables using a single statement. In particular, notice the string containing the space: it is necessary to ensure that the two numbers don't run together. It is possible to output complex numbers, matrices, strings, and vectors in addition to the basic types. One can embed carriage returns in the output using the '\n' character

```
In[1]:
Real a = 1.4, b = -1.3;
In[2]:
cout << ''The Answer is \n'' << a << '' '' << b;
```

[3]Instantiation is a synonym for creation

```
  The Answer is
  1.4 -1.3
In[3]:
```

Consistent with C++, a carriage return is *not* automatically placed after each stream output statement involving `cout`. This means that executing

```
for (i = 1; i <= 5; i++)
  cout << i;
```

produces

```
12345
```

The `cout` stream is meant for output. Accordingly, it is buffered in 1024 character blocks. Another stream, `cerr`, which is line buffered is also usable in the shell. This stream is useful for error messages.

It is also possible to perform stream I/O with files. This is accomplished using the `Fstream` and `ios` classes

```
class Fstream {
public:
  Fstream();
  Fstream(const String &str, Integer ioflag);
  ~Fstream();

  Void open(const String &str, Integer ioflag);
  Void close();
};

class ios {
public:
  static Integer in;
  static Integer out;
};
```

Notice that the `ios` class contains a new type of `static` data member. Static data members do not belong to a particular object. Rather, all instantiations of the class access the same value. Since the static member is not tied to a particular object they need not be accessed using the *object.member* syntax. Instead, they can be referred to using a *qualified name* with the syntax

class-name :: *static-data-member-name*

Therefore, to access the static member out in class ios one can type ios::out. Note that it *would be* possible to use the usual *object.member* syntax if an ios object could be instantiated. However, since the ios class has no constructors that is not possible in the SUPERCODE shell. In this case, the qualified name must be used. Note that it is possible to have static member functions as well. They can be called using the qualified name syntax

class-name :: *static-function-member-name*(*argument-list*)

The Fstream constructors allow the following shell code to be executed. This fragment

```
In[1]:
Real a = 1.4, b = -1.3;
In[2]:
Fstream fio(''test.dat'', ios::out);
In[3]:
fio << a << '' '' << b << ''\n'';
In[4]:
```

would result in a file called "test.dat" containing the text data

```
1.4 -1.3
```

The declaration of fio creates (if necessary) the file "test.out", cleans out all of the current contents (if necessary), and opens the file for writing. The stream output statement actually store the data in the file. Additional statements involving fio would append data to the file. The file remains open until the file closing member function

```
fio.close();
```

is issued.

It is possible to re-open a file using the file opening member function

```
fio.open(''test.dat'', ios::in);
```

The open member function takes the same arguments as the Fstream constructor. Notice that the above statement opens test.dat for *reading*. It is now possible to re-read the contents of the file by "re-directing" the stream operators:

```
In[1]:
Real a1 = 0.0, b1 = 0.0;
In[2]:
fio >> a1 >> b1;
In[3]:
a1; b1;
  1.4
  -1.3
In[4]:
```

Currently, it is not possible to read 3-D array, matrix, string, or vector variables.

The shell supports a wide variety of formatting options. See a C++ book for details.

Chapter 3

Writing Modules

3.1 Introduction

The computational part of the SUPERCODE is divided into a number of *modules*. Each of these modules contains data and functions that are related by a single unifying abstraction. For instance, a module named *Const* might contain the values of useful mathematical and physical constants. Or, perhaps, a module named *TFCoil* might contain data and functions relevant to calculating characteristics of the ITER toroidal field system.

There are two main reasons for the division into modules. First, this partitioning helps reduce the complexity inherent in a computer program the size of the SUPERCODE. We expect that, eventually, the SUPERCODE will consist of hundreds of thousands of statements. Most people will not have the time, or the inclination, to understand all of the code. Modules allow people to consider the code in small chunks that can reasonably be understood. The second reason for creating modules is practical. Several physicists and engineers from various institutions will be simultaneously working on the SUPERCODE project. Modules allow parts of the code to be designed and revised independently.

The act of partitioning the code does not guarantee that we will succeed in our two goals. For instance, suppose that a user of the *TFCoil* module made explicit assumptions regarding the details of the algorithm used to compute TF coil stresses. Then, changes to that algorithm would require changes elsewhere in the code. This coupling, which can be quite subtle, serves to increase

complexity and disallow independent development.

It is important to recognize that we are not stating that all dependencies between modules are undesirable. Clearly, some coupling between modules in unavoidable. However, the *type* of coupling is the issue. It is not unreasonable to expect that another engineering module might be concerned with the values of the TF coil stresses. However, other modules should not care whether the TF stresses are computed using a simple scaling algorithm or an elaborate 3D solve.

In order to figure out how to intelligently handle the coupling between modules, it is useful to think of a module as having two parts: an *interface* and an *implementation.* The interface of a module describes how it appears to an outside observer or, put another way, how it behaves. The implementation of a module specifies the mechanisms that achieve the desired behavior. Consider the TF coil module. The interface consists of the data needed to perform its calculations (e.g., the shape of the TF coil, the magnetic field, etc), the functions one needs to call to do the calculations (e.g., `calcStresses`), and the results from the calculations (e.g., the stresses, nuclear heating, etc). The implementation consists of the specific algorithms used to calculate the results.

Eliminating the dependencies we are worried about is achieved by *encapsulating* the details of a module's implementation. This, in turn, can be accomplished by making sure that users always use function calls to calculate things. Here, users of the *TFCoil* module would call a routine `calcStresses` to get the TF coils stresses. As we mentioned, this function might compute the stresses using a simple scaling method or an elaborate 3D solve. However, as long as the interface to this calculation (i.e., the name of the function) doesn't change, the source for modules that use the routine need not be changed.

Function calls provides a way to encapsulate module implementations. However, this alone is not sufficient. First, modules will invariably need to store data in order to perform their calculations. Furthermore, some of this data will be peculiar to a specific algorithm. If other modules access this data, the undesirable coupling returns. In addition to data, module writers will probably want to define functions to implement algorithm-specific calculations. What is to stop users from calling these routines?

It seems that there needs to be two classes of information in a module: *public* data and functions that are supposed to be accessed by users of the module and *private* data and functions that are for internal use only. In the *TFCoil* module, an example of public data might be the desired toroidal field

at the plasma center (an input) or the stress in the inner TF leg (an output). The routine `calcStresses` would be a public function callable by users of the module. Coefficients used for scaling calculations might be private data members while a routine that performs part of the stress calculation might be a private function.

In this section, we have introduced the concept of modules and the idea of having public and private data and function module members. Up to this point the discussion has been quite general. In the balance of this chapter we will provide specifics as to how to write such modules for incorporation into the SUPERCODE. We will also describe how to access the data and routines making up a module from the SUPERCODE shell.

3.2 Writing the Interface

The first step in developing a module is creating the interface. This means determining what the public data and functions are and what the private data and functions are. Once this is done, it is necessary to pass this information on to other module writers since they cannot use a module unless *they* know what the interface looks like.

For the SUPERCODE project, we have chosen to formalize this process of exchanging information by requiring that all developers supply a *module description file* specifying the interface for each module they contribute. This requirement has a number of desirable features

1. *Reliability.* Since module description files have a particular format (soon to be described) they can be read by a computer program which then generates the necessary code and common block declarations required to allow modules to talk to each other. This program also can generate the code required to give the shell access to the module members. This automatic generation of interface code reduces programming errors and makes handling changes in the interface less tedious for the user.

2. *Language-independence.* The required interface code depends on the programming language. Since this code is generated automatically, it is possible to have some modules written in FORTRAN and some modules written in C++ while, at the same time, hiding this detail from the user.

3. *Portability.* The required interface code may depend on the computing platform. Since this code is generated automatically these system-specific issues can be hidden from the user.

4. *Documentation.* A properly-written module description file can serve as documentation for a module. Therefore, no secondary documents need be written.

We believe these features outweigh the need to learn how to correctly format the module description file.

3.2.1 Module Description File Naming Conventions

By convention, module description files are named *ModName.mod* where *ModName* is a capitalized one-word label for the module. For example, the module description file name for the *TFCoil* module would be *TFCoil.mod.*

3.2.2 Module Description File Format

A module description file has the basic syntax

```
module module-name-with-info {
  module-element
  module-element
    ⋮
  module-element
}
```

where the *module-name-with-info* specifies information about the module as a whole and *module-element* is one of

- The access specifier "`public:`".
- The access specifier "`private:`".
- A data member declaration.
- A function member declaration.
- A module import declaration. (This tells what modules this module uses data or functions from.)

- An exception declaration. (This describes a warning, error, or fatal that this module can generate.)

There may be any number of *module-elements* and they may be listed in any order. Also, comments of the type discussed in Section 2.3.2 can be used anywhere.

A concrete example of these elements is contained in the module description file below which gives the interface for the hypothetical module *Example*:

```
1   // Example.mod
2
3   module Example - ``A sample module'' {
4     Uses Const, Sys;
5     Errors badGridSize
6                         - ``Illegal number of grid points.''
7                         - ``Grid size < 0 was passed to resizeGrid.'';
8   public:
9     fortran Void ctor()
10                        - ``Constructor for the Example module.''
11                        - ``Initializes grid and allocates storage.'';
12    fortran Void dtor()
13                        - ``Destructor for the Example module.'';
14
15    fortran Void resizeGrid(const Integer &newNGPTS)
16                        - ``Resets the grid to the specified size.'';
17
18    fortran Integer ngpts()
19                        - ``Returns the current number of grid points.'';
20    RealVector gridVals
21                        - ``Grid location values.''
22                        - ``By default, the grid is rectangular.''
23                        - ``This variable is for reading ONLY.'';
24
25  private:
26    Integer cngpts
27                        - ``The current number of grid points.'';
28    fortran Void calcGrid(RealVector &gvals)
29                        - ``Computes grid location values.'';
30  }
```

This module description file declares four public functions (`ctor`, `dtor`, `ngpts`, and `resizeGrid`), one public data member (`gridVals`), one private function (`calcGrid`), and one private data member (`cngpts`). It also indicates that is module uses data or functions from the `Sys` and `Const` modules. Finally, an error exception `badGridSize` is declared. We will now examine the syntactic elements of the file in more detail.

Module Name Specifier

Line 3 of the sample file shows an example of the *module-name-with-info* element:

```
Example - ''A sample module.''
```

The first word specifies the name of the module which should match the first part of the file name. The next part is a construct that gives information that is to be stored in the driver-shell's run-time database. In this case, the string "A sample module." is associated with the name "Example". Eventually, it will be possible to access this information using shell commands but that capability has not been implemented yet. As the sample file shows, this pattern of the name of an object followed by an information string (or strings) appears many times. In each case, the string(s) represent descriptive information about the object that is to be stored in the shell database.

As the sample file shows, it is possible to associate more than one line of information with an object. This is accomplished by simply appending more strings, separated by dashes, to the name. For example, a more detailed description of the module could be written

```
Example - ''A sample module.''
        - ''This module manages a one-dimensional rectangular grid array.''
```

By convention, the first string gives a brief description of the element. Subsequent strings should give more detailed information. In addition to supplying the information to the shell database, the information strings should also provide documentation for users of the module.

Access Specifiers

The access specifier "`public:`" indicates that subsequent function and data members are public. Accordingly, these members can be used by other modules. This remains the case until a "`private:`" access specifier is seen. At that

point, all members are deemed private and they *cannot* be accessed by other modules. There can be any number of groups of public or private members; however, by convention, public members are listed first.

Data Member Declarations

As the sample file indicates, data members are declared in a manner very similar to variables in the shell (see Section 2.3.3). However, for compatibility with FORTRAN, the types of variables that can be declared is limited. In particular,

- The only allowed types are `Complex`, `Integer`, `Real`, `ComplexArray3`, `IntegerArray3`, `RealArray3`, `ComplexMatrix`, `IntegerMatrix`, `RealMatrix`, `ComplexVector`, `IntegerVector`. `RealVector`, `String`, and `Subroutine`.
- No constant or reference variables are allowed.
- Variable names must start with a letter.
- Variables cannot be initialized in the module description file.

Note that data members can be initialized using a mechanism to be described shortly. Moreover, in the future, the prohibition on initializing data members in the module description file may be relaxed. By convention, data member names should start with a lower case letter.

It is possible to declare several data members at once. For example

```
Integer i
          - ''Variable i description'',
        j
          - ''Variable j description 1''
          - ''Variable j description 2'',
        k,
        l
          - ''Variable l description'';
```

Note carefully the placement of commas and the semicolon. The system is relatively easy once you get the hang of it.

Function Member Declarations

As with data members, the function member declaration syntax is very similar to that of the shell language. Again, for compatibility with FORTRAN, there are limitations on the sorts of functions that can be declared. These are

- Functions can only return types `Integer`, `Real`, and `Void`.
- The return type cannot be a reference or a constant.
- If the function is actually implemented in FORTRAN, the `fortran` keyword must be placed in front of the declaration. By default, it is assumed that functions are written in C++.
- The only allowed argument types are `Complex`, `Integer`, `Real`, `ComplexArray3`, `IntegerArray3`, `RealArray3`, `ComplexMatrix`, `IntegerMatrix`, `RealMatrix`, `ComplexVector`, `IntegerVector`, `RealVector`, `String`, and `Subroutine`.
- All arguments, other than those of type `Subroutine` must be declared as references. It is possible to pass constant references as arguments (i.e., `const Real &`).
- Function names must start with a letter.
- There cannot be two members with the same name.

The sample module description file gives a number of examples of function member declarations. By convention, function member names should start with a lower case letter.

Module Import Declarations

The modules that the current module expects to get data or call functions from must be listed in a module imports declaration. The form of this declaration depends on whether the module is written in FORTRAN or C++. FORTRAN modules use the keyword `Uses` while C++ modules use the keyword `Inherits`. By convention, the module imports declaration is the first module element.

Exception Declarations

Since the SUPERCODE is interactive, it is important for modules to perform error checking. If a problem is found, the module can handle the error using an exception declared in the module description file. Three types of exceptions can be generated: warnings, errors, and fatals. These correspond to the exceptions described in Section 2.1.

An exception declaration essentially associates an exception name with a message. In the sample file, the exception `badGridSize` is associated with a relevant error message. Exception declarations have the same syntax as data member declarations except that the "type" is either `Warnings`, `Errors`, or `Fatals`. The information strings for exceptions are treated slightly differently though. In the case of exceptions, they represent the text of the message that is to be printed by the shell upon hearing about the exception. Accordingly, at least one information string for each exception must be supplied. Only the first three strings will actually appear in the error message but more can be provided in the module description file for documentation purposes. By convention, exception declarations are listed in order of increasing severity right after the module imports declaration. Also, exception names start with a lower case letter.

In a later section we will show how to actually raise exceptions.

Special Functions

Modules include two special function members called `ctor` and `dtor`. The `ctor` function is known as the *constructor* for the module and the `dtor` function is known as the `destructor` for the module. When the SUPERCODE is started up, the shell calls the `ctor` functions for all of the modules. When the `quit` directive is executed, the `dtor` functions are called in the reverse order.

Every module that stores information should define a `ctor` function in the module description file. Then, inside the `ctor` function, initialization functions should be performed with the goal of making sure that the module is in a consistent state at the start. Typical activities carried out in a `ctor` include initializing dimensions, resizing matrices and vectors, filling in data, or perhaps opening up files.

The use of the `dtor` function is much less common. However, if there are any clean-up activities that need to be performed when the code shuts down

they should be done in the `dtor`. Storage is deallocated automatically so that is not a worry. However, closing or writing a file may be something a `dtor` might do.

Modules that do not need the services of a constructor or destructor need not define one: they will be generated automatically.

3.2.3 MGen

We alluded to the fact that the module description file could be read by a computer program which would produce the code needed to allow modules and the shell to communicate. The name of this program is *MGen* and it can be executed on our sample module description file by simply typing

```
MGen Example.mod
```

When *MGen* gets done reading the module description file, you will notice that four files have appeared in your directory:

- *Example.i* This file contains common blocks and definitions required to allow *Example*'s FORTRAN member functions to access the *Example* module's data and function members along with the public data and function members of the *Sys* and *Const* modules that *Example* imports.
- *Example.h* This file contains definitions required to allow *Example*'s C++ member functions and the shell to access *Example* module members.
- *ExampleMBase.m* This file contains FORTRAN glue code required to allow the shell to access *Example* module members.
- *ExampleCBase.cc* This file contains C++ glue code required to allow the shell to access *Example* module members.

Generally, module writers should not be concerned with the contents of these files. Moreover, these files contain some of the yuckiest looking code ever written so it's best to leave these alone.

3.3 Writing the Implementation

Having shown how to write a module interface, it is now necessary to show how to write the module implementation. The details of this procedure depend on

the programming language. We will first show how to write an implementation in FORTRAN and then how to write one in C++. Note, however, that it is not necessary for a module to be written entirely in one language! Some functions can be written in FORTRAN for efficiency and some can be written in C++ for convenience.

3.3.1 Using FORTRAN

FORTRAN module implementations are initially read by a pre-processor called *MPPL*. MPPL converts the implementation to FORTRAN77 code which can then be compiled. Files that are designed to be read by MPPL must have the suffix `.m` while pure FORTRAN files have the suffix `.f`.

MPPL supports a number of enhancements to FORTRAN including

- Free format input.
- Shell-style // comments.
- Sophisticated macro capabilities.
- Simplified looping and selection statements.

These capabilities are described in detail in the MPPL manual [8]. Note that module writers do not *have* to use these extensions: with a few exceptions, modules can be written in pure FORTRAN77. However, MPPL's enhancements make FORTRAN a much more pleasant language to use.

In the balance of this section, we will show, by means of our sample module *Example*, how to write a FORTRAN implementation. For simplicity, we assume that all of the source code for the module resides in the file *Example.m*.

Near the top of *Example.m* (before any functions or subroutines) it is necessary to have the line

```
include Example.i
```

Note that there are no quotes around the file name. This directive causes the contents of the *Example.i* definitions file generated by *MGen* to be read.

Now we can begin writing the member functions. Let us start with the constructor. It is written

```
subroutine Example_ctor
BeginExampleMember()

  call resizeGrid(8)
  return

EndMember
```

Part of this looks like standard FORTRAN and part doesn't. The first difference is the name of the subroutine. The "`Example_`" is prepended to `ctor` here to indicate that this is a member of the module *Example*. The next line is a macro that causes the common blocks holding the data members of the *Example* module to be included. All calls to the `BeginExampleMember`[1] macro must be balanced by a call to the `EndMember` macro which replaces the subroutine's end statement. In between these macro calls is fairly standard FORTRAN code that initializes the grid to a size of 8.

At this point it is pertinent to discuss references to module members. For the most part, it is possible to refer these members using the name as written in the module description file (e.g. `resizeGrid`). The only exception to this rule is when a public member of an imported module contains a member with the same name as a member of the present module. An example of this situation is the `ctor` function which is a member of all modules. If one simply included a line

```
call ctor
```

in a member function, it is not clear whether we are supposed to use the `ctor` in *Sys*, *Const*, or *Example*. Therefore, MPPL will generate an error saying that the call is ambiguous. The correct way to do it is to prepend the module name:

```
call Sys_ctor
```

This say that we are calling *Sys*'s `ctor`. Note that all members can be referred to in this way (e.g., `Example_resizeGrid`, `Example_cngpts`, etc). However, prepending the module name is not necessary unless the reference would otherwise be ambiguous.

Having written the constructor, we move to the destructor

[1]Note: if the module name is `Mod`. the macro name is `BeginModMember`.

```
subroutine Example_dtor

  return

end
```

Since this routine doesn't do anything, we could have omitted its declaration from the module description file. Then it would have been written by *MGen*. Notice that there is no BeginExampleMember/EndMember macro pair for this subroutine. This is not necessary because the routine does not access any of the module's members. Leaving off these unnecessary macro calls will decrease compilation time and code size so, whenever possible, they should be eliminated.

The resizeGrid member shows how to handle arguments

```
subroutine Example_resizeGrid(newNGPTS: integer)
BeginExampleMember(Sys)

  if (newNGPTS < 0) then
    call except(badGridSize)
  endif
  cngpts = newNGPTS
  call rvResize(RealVector(gridVals), newNGPTS)
  call calcGrid(RealVector(gridVals))

  return

EndMember
```

Notice that the type of the argument is declared using the syntax

arg-name : *type*

where the *type* maps to the corresponding type from the shell or module description file according to Table 3.3.1. No const keywords appear here since FORTRAN has no mechanism for ensuring that const items do not get changed. It is up to the programmer to ensure that these are read-only. Also, it is not necessary to place the & indicating that references are passed since this is always the case with FORTRAN. In addition, see that the BeginExampleMember macro takes an argument "Sys". This argument

Table 3.1: Shell/Module Description File Type Mapping to MPPL Types

Shell/MDF Type	MPPL Type
Complex	complex
Integer	integer
Real	real
ComplexArray3	complexarray3
IntegerArray3	integerarray3
RealArray3	realarray3
ComplexMatrix	complexmatrix
IntegerMatrix	integermatrix
RealMatrix	realmatrix
ComplexVector	complexvector
IntegerVector	integervector
RealVector	realvector
Subroutine	subroutine

specifies that the routine will make use of variables and/or functions from the imported Sys module. If the routine also made use of variables from the Const module the macro would read BeginExampleMember(Const,Sys).

The resizeGrid member is supposed handle changes in the grid size. First, it checks the desired grid size to make sure it is positive. If it isn't, it calls the exception raising subroutine except. This subroutine expects one argument: the label of the exception. If the exception is a warning, an message is printed and execution proceeds on the line following the except call. Errors and fatals, on the other hand, do not return. If the new grid size is acceptable, the private variable cngpts is updated and the routine rvResize (a member of the *Sys* module, see *Sys.mod*) is called. This routine, which has the declaration,

```
Void rvResize(RealVector &vec, const Integer &nm)
```

resizes gridVals to have newNGPTS elements. Note that the "RealVector(···)" is required. This is a macro call that ensures that all of the correct information is passed to a function expecting a RealVector & as an argument. There are corresponding macro calls for ComplexMatrix, ComplexVector, etc. Be sure

to use this macro or else disaster will follow since FORTRAN does no type-checking. Finally, the private member function `calcGrid` is called to actually compute the grid values and store them into `gridVals`.

To resize Array3 variables, use the routines `ca3Resize`, `ia3Resize`, and `ra3Resize`. These take three dimensions. For example,

```
call ca3Resize(ComplexArray3(x), 3, 5, 7)
```

resizes a `ComplexArray3` variable `x` to be $3 \times 5 \times 7$. Be sure to use the "`ComplexArray3(···)`" macro.

The member `ngpts` is a very simple function

```
integer function Example_ngpts()
BeginExampleMember()

  return(cngpts)

EndMember
```

Making `ngpts` a function means that users of the module can look at the number of grid points but cannot modify it haphazardly. This is the best approach if there needs to be side effects associated with changing a variable. Note that the syntax for returning answers from functions is different than standard FORTRAN. The normal approach of assigning to the function name will not work in SUPERCODE modules.

We finally come to the `calcGrid` routine which actually does some arithmetic

```
subroutine Example_calcGrid(gvals:  realvector)
BeginExampleMember(Const)
  integer i

  do i = 1, rows(gvals)
    gvals(i) = twoPi * float(i - 1) / float(rows(gvals))
  enddo

  return

EndMember
```

The constant `twoPi` comes from the *Const* module. This routine illustrates a few important points. First, all of the matrix and vector types know what their dimensions are. The number of rows of a vector[2] can be accessed using the `rows` macro while the number of rows or columns of a matrix can be accessed using the `rows` or `cols` macros. The second important point is that, while `RealVector` types must be handled specially when passing them as arguments, they can be manipulated in arithmetic expressions exactly like FORTRAN arrays.

Some of the syntax in this section probably seems strange. However, a little practice should make writing modules in FORTRAN a snap.

3.3.2 Using C++

Writing modules in FORTRAN requires the use of a number of MPPL extensions. This is because FORTRAN77 has little inherent support for modularity or information hiding.[3] C++, on the other hand, includes these elements by default. Moreover, you will see that the syntax is extremely close to that of the shell language.

In the balance of this section, we will rewrite our sample module in C++. The only changes one needs to make to the module description file is the deletion of the `fortran` directives in front of the function member declarations and the replacement of `Uses` with `Inherits` in the import declaration line.

For simplicity, we assume that all of the source code for the module resides in the file *Example.cc*[4].

Near the top of *Example.cc* (before any functions or subroutines) it is necessary to have the line

```
#include ''Example.h''
```

This directive causes the contents of the *Example.h* definitions file generated by *MGen* to be read.

Now we can begin writing the member functions. Let us start with the constructor. It is written

[2]Vectors are assumed to be column vectors.

[3]The new FORTRAN90 standard incorporates many of the features we have had to implement with MPPL including modules and public and private member functions.

[4]The suffix `.cc` is assumed for C++ files.

```
Void Example::ctor()
{
  resizeGrid(8);
}
```

The name of the module is connected to the name of the member with a double-colon `::`. This replaces the underscore of FORTRAN. There are no common blocks in C++ so no `BeginMember/EndMember` pair of macros is required. The body of the function consists of a call to `resizeGrid` which initializes the grid to a size of 8.

The ambiguity resolution rules for C++ are the same as for FORTRAN except that member references have the forms `Example::resizeGrid`, etc. Again, prepending the module name is not necessary unless the reference would otherwise be ambiguous.

As before, the destructor is empty

```
Void Example::dtor()
{
}
```

Since this routine doesn't do anything, we could have omitted its declaration from the module description file. Then it would have been written by *MGen*.

The `resizeGrid` member shows how to handle arguments

```
Void Example::resizeGrid(const Integer &newNGPTS)
{
  if (newNGPTS < 0)
    except(badGridSize);
  cngpts = newNGPTS;
  rvResize(gridVals, newNGPTS);
  calcGrid(gridVals);
}
```

Notice that the formal argument declarations in C++ match exactly the ones in the module description file. In addition, the "`RealVector(···)`" macro is not required in C++ since this language knows how to pass matrix and vector types.

The member `ngpts` is very straightforward

```
Integer Example::ngpts()
{
  return(cngpts);
}
```

as is calcGrid:

```
Void Example::calcGrid(RealVector &gvals)
{
  Integer i;

  for (i = 1; i <= rows(gvals); i++)
    gvals(i) = (twoPi * (i - 1)) / rows(gvals);
}
```

Again, it is important to note the the matrix and vector types work just like FORTRAN arrays (including indexing from 1). Moreover, the rows and columns can be accessed using `rows` and `cols`.

3.4 Recompiling the Code

The SUPERCODE compile and link process is controlled by a powerful UNIX utility called *imake*. *Imake* is a front-end for the well-known UNIX *make* program. The primary advantage of *imake* is that it provides a way to hide all of the system-dependent information required to build a program from the user. As a result, Imakefiles (the *imake* equivalent of a makefile) are extremely short, even for complicated programs like the SUPERCODE.

After writing the interface and implementation, the next step in adding a module to the SUPERCODE is revising the Imakefile. The SUPERCODE Imakefile is *SuperCode/SC/Imakefile*. We will now discuss how to modify this file to add FORTRAN and C++ modules.

3.4.1 Adding a Simple FORTRAN Module

The Imakefile changes required to add a module consisting of a module description file and a *single* MPPL source file is shown below

```
SetDefaultOptimization(-O3)
```

```
#ifdef InObjectCodeDir

SourceFileDependencies()
ModFileDependencies()

SimpleCCModuleTarget(Const)
SimpleCCModuleTarget(Sys)
SimpleMpplModuleTarget(Example)

all::
→  $(RM) modlist.h
→  EchoModuleEntry(Const)
→  EchoModuleEntry(Sys)
→  EchoModuleEntry(Example)

AllTarget(SuperCode)
DependTarget()

OBJS = SuperCode.o ModConst.o ModSys.o ModExample.o

SuperCode.o:  modlist.h

SuperCode :  $(OBJS)
→  $(CCC) -Bstatic -o SuperCode $(OBJS) $(PROJECT_STDLIBS)

clean::
→  $(RM) SuperCode

InstallProgram(SuperCode,$(PROJECT_BINDIR))

Example.o: Const.i Sys.i

#else

MakeInObjectCodeDir()

#endif
```

Originally, this Imakefile built the SUPERCODE using only the *Const* and the *Sys* modules. The modifications required to add our sample module *Example* are shown in **bold face**. The arrows (→) indicate that these lines must begin with a TAB. Basically, one must simply specify the name of module in a few places and supply a line giving the include files for imported modules (it is not necessary to include *Example.i* on the line).

3.4.2 Adding a Simple C++ Module

If a module consists of a module description file and a single C++ source file, the modifications to the Imakefile are almost identical to those required to add a FORTRAN module. If our module *Example* were written in C++, the Imakefile above would become

```
SetDefaultOptimization(-O3)

#ifdef InObjectCodeDir

SourceFileDependencies()
ModFileDependencies()

SimpleCCModuleTarget(Const)
SimpleCCModuleTarget(Sys)
SimpleCCModuleTarget(Example)

all::
→  $(RM) modlist.h
→  EchoModuleEntry(Const)
→  EchoModuleEntry(Sys)
→  EchoModuleEntry(Example)

AllTarget(SuperCode)
DependTarget()

OBJS = SuperCode.o ModConst.o ModSys.o ModExample.o

SuperCode.o:  modlist.h

SuperCode :  $(OBJS)
```

```
→  $(CCC) -Bstatic -o SuperCode $(OBJS) $(PROJECT_STDLIBS)

clean::
→  $(RM) SuperCode

InstallProgram(SuperCode,$(PROJECT_BINDIR))

Example.o: Const.h Sys.h

#else

MakeInObjectCodeDir()

#endif
```

where, as before bold face indicates the required changes.

3.4.3 Adding a more Complex Module

It is not necessary for the source for a module to reside in a single source file. If this is the case, a slightly different set of modifications to the Imakefile is required. Consider, for instance, a new module called *Optimize*. This module is a front-end for the VMCON constrained optimization package. It consists of a module description file *Optimize.mod*, a C++ source file *Optimize.cc*, and a pure FORTRAN source file *VMCON.f*. The Imakefile changes required to add *Optimize* to the SUPERCODE (which already includes the *Example* module) are shown below

```
SetDefaultOptimization(-O3)

#ifdef InObjectCodeDir

SourceFileDependencies()
ModFileDependencies()

SimpleCCModuleTarget(Const)
SimpleCCModuleTarget(Sys)
ModuleTarget(Optimize,Optimize.o VMCON.o)
SimpleCCModuleTarget(Example)
```

```
all::
→  $(RM) modlist.h
→  EchoModuleEntry(Const)
→  EchoModuleEntry(Sys)
→  EchoModuleEntry(Optimize)
→  EchoModuleEntry(Example)

AllTarget(SuperCode)
DependTarget()

OBJS = SuperCode.o ModConst.o ModSys.o ModOptimize.o ModExample.o

SuperCode.o:  modlist.h

SuperCode :  $(OBJS)
→  $(CCC) -Bstatic -o SuperCode $(OBJS) $(PROJECT_STDLIBS)

clean::
→  $(RM) SuperCode

InstallProgram(SuperCode,$(PROJECT_BINDIR))

Optimize.o: Optimize.h
Example.o:  Const.h Sys.h

#else

MakeInObjectCodeDir()

#endif
```

where, as before **bold face** indicates the required changes. Basically, one must simply specify the name of the module, the object files that make up the module, and the include files the objects depend on. Unlike the simple case, *all* include files must be specified (not just the imports).

3.4.4 Recompiling

Anytime the Imakefile is changed, it is necessary to issue the command

```
make Makefile Makefiles
```

This tells *imake* to reconfigure its files for the new module. To compile and link the whole code simply type

```
make
```

Don't worry about the details, *imake* knows what to do.

If an error during one of the compiles, note the name of the file that had the problem. Then look in the *SUN4* directory and you will see a bunch of files with the suffix `.err`. Open up the file that has the correct file prefix in order to see the error. Upon fixing the problem, simply type `make` again.

If the build finishes without problems, a brand-new version of the SUPERCODE should appear as *SuperCode/SC/SUN4*. It can be executed by typing `SuperCode` in the *SuperCode/SC* directory if you include `./SUN4` in your path (this is done in *cshrc.scode*.

3.4.5 Compiling with Debugging Information

By default, *imake* compiles optimized code. However, sometimes it is necessary to make use of a debugger such as *dbx*. In order to do this simple change the line

```
SetDefaultOptimization(-O3)
```

in the Imakefile in the *SC* directory to

```
SetDefaultOptimization(-g)
```

Then, type "`make Makefile Makefiles`". At this point, you must simply recompile the files you wish to debug to generate debugging information. To switch back to using optimized code, change the `SetDefaultOptimization` macro back. See the manual for whatever debugger you are using for debugging instructions.

3.5 Accessing Modules From the Shell

Data and function members of a module can be accessed from the shell. The safest way to do this is to prepend "`ModName::`" (where `ModName` is the name of the module) to the member names. Using this convention, we can access members of our sample module *Example* as follows

```
In[1]:
Example::ngpts();
  8
In[2]:
Example::gridVals;
  {{ 0 },
   { 0.785398 },
   { 1.5708 },
   { 2.35619 },
   { 3.14159 },
   { 3.92699 },
   { 4.71239 },
   { 5.49779}}
In[3]:
Example::resizeGrid(-1);
### Error 81: Illegal Number of Grid Points.
        Grid size < 0 was passed to Example::resizeGrid.
        File "Interactive"; Line 3 # caused the exception.
        File "ExampleCBase.cc"; Line 198 # generated the exception.
In[4]:
Example::resizeGrid(4);
In[5]:
Example::gridVals;
  {{ 0 },
   { 1.5708 },
   { 3.14159 },
   { 4.71239 }}
In[6]:
```

A more elaborate example that exercises the *Optimize* module is contained in *opt.sc*.

Under most circumstances, it *is* possible to omit the module name qualification in front of the member name. For instance,

```
In[1]:
ngpts();
  8
In[2]:
```

calls the desired function `Example::ngpts()`. The reason the qualification can sometimes be neglected is because of a *search list* maintained by the shell.

When an un-qualified name is typed, the shell first attempts to find this name in its database given the current scope level. If this search fails, the shell then sequentially goes through the search list looking for module qualifications that would result in a successful lookup. If, a *single* match is found, the shell assumes that the variable desired *implicitly* possessed the required qualification. In the example above, the shell found that no `ngpts` was defined at global scope and, moreover, that one, and only one module possessed a member by that name. Therefore, the function was called. If a version of `ngpts` existed at global scope, that version would have been called instead. If two modules had members named `ngpts` an error exception would have been generated. In this case, the module qualification must be used so that the shell knows which member is desired. An example of the issues involved is contained in the example:

```
In[1]:
Integer ngpts() { return 7; }
In[2]:
ngpts();
  7
In[3]:
Example::ngpts();
  8
In[4]:
```

By default *MGen* adds all modules to the search list. Therefore, one can usually omit the qualification unless trying to access a name defined simultaneously in two modules.

Note: you can also use the `list` and `about` functions described in Section 4.2.2 to find out the structure of Modules.

Chapter 4

Standard Modules

4.1 The Consts Module

One often needs to use mathematical and scientific constants such as π and μ_0 in one's code. According, we include a very simple, data-only, module giving various useful values. See the module description file "Consts.mod" to see the current set of available constants.

4.2 The Sys Module

The *Sys* module gives an interface to various system functions controlled by the shell. The functions fall into a number of categories:

- Array resizing.
- Database browsing.
- Date and time information.
- Error trapping.
- Timing.
- Version information.

Let us describe these functions in more detail.

4.2.1 Array Resizing

It is possible to resize a RealArray3 object using the function

```
Void ra3Resize(const RealArray3 &a, const Integer &dim1,
  const Integer &dim2, const Integer &dim3)
```

where dim1, dim2, and dim3 are the new dimensions. To resize ComplexArray3 objects use ca3Resize and to resize IntegerArray3 objects use ia3Resize.

To resize matrices use cmResize, imResize, and rmResize. These routines are similar to ra3Resize except, of course, they only expect two dimensions.

To resize vectors use cvResize, ivResize, and rvResize. These routines are similar to ra3Resize except, of course, they only expect one dimension.

All of these routines can be called from Fortran.

4.2.2 Database Browsing

It is possible to find out about classes and variables while running the SUPERCODE using the list abd about functions. The list function is used to find out the names of symbols currently defined in the SuperCode. This function has the prototype:

```
Void list(const String &);
```

The string should be a regular expression. (See any UNIX book for a discussion of what these are.) The list function outputs all symbols in the SUPERCODE database that match the specified regular expression. For example, try the following

```
list(".*"); // Lists all global symbol names.
list("a.*"); // Lists all global symbols starting with an 'a'.
list(".*::"); // Lists all type and module names.
list("Array3::d.*"); // Lists all members of the Array3 class
                     // starting with a 'd'.
list(".*::ctor"); // Lists all the 'ctor' functions in all classes.
```

A complementary function is about. This has the prototype

```
Void about(const String &);
```

The String in this case is a symbol name possibly found by `list`. For names that represent variables, the type and a description of the variable are printed. For function names, the return type, argument types, and a description are printed. For classes, the types and descriptions about all members is printed. For example, try typing

```
about("_i"); // Tells about the global variable '_i'.
about("exp"); // Tells about the function 'exp'.
about("Complex"); // Tells about the Complex class.
about("Vector::length"); // Tells about the member function.
about("ctor"); // Tells about all 'ctor' member functions.
```

Remember, the string passed to about is not a regular expression.

4.2.3 Date and Time Information

It is possible to get a string representation of the current date and time by calling the function

```
Void dateTime(String &d)
```

where `d` is a string to store the information.

4.2.4 Error Trapping

By default, the shell catches exceptions such as floating point errors, etc. Sometimes, for debugging purposes, it is best to allow the shell to dump core when an error occurs. This can be done by calling the function

```
Void errorTrap(const Integer &onoff)
```

with the boolean value `False`. To turn error trapping back on, call `errorTrap` with `True`.

4.2.5 Timing

One often wants to gather timing information. This can be accomplished by first calling the routine `timerOn()` and then, after the task that is being timed, the routine `elapsedTime()`, which returns a the time, in seconds, since the call to `timerOn`.

4.2.6 Version Information

It is possible to get a string representation of the shell version by calling the function

```
Void shellVersion(String &d)
```

where d is a string to store the information. The routine `applVersion` returns the application version information.

4.3 The Plot Module

The ability to make plots of data is an important capability for an interactive scientific code. Therefore, let us describe a plotting facility, implemented in the *Plot* module and shell code, supporting X–Y plots, contour plots, and 3–D surface plots. This module is based on the PLPLOT library, developed by LeBrun, *et al.* [9].

The goals for plotting facility's user interface are simplicity, expressiveness, flexibility, and compatibility. Our interface should encourage use by making it easy to make a simple plot. On the other hand, it should allow for the construction of complicated plots with a variety of different characteristics. Finally, we want the interface to be consistent with C++ syntax (i.e., no changes to the Shell language required).

4.3.1 Basic X–Y Plots

The simplest sort of plot involves graphing a series of ordinate (Y) values against a series of abscissae (X) values. We call this an X–Y plot. All X–Y plots are made using a call to the routine `xyPlot`. This routine has the prototype

```
Void xyPlot(const RealMatrix &y, const RealMatrix &x, option-list)
```

where

- y is an $N \times NX$ matrix holding NX ordinate arrays of length N.
- x is either a vector of length N or a $N \times NX$ matrix; x holds either one or NX abscissae arrays.

The *option-list* refers to an optional set of additional arguments that we will consider later.

The simplest X–Y plot involves plotting one vector versus another:

```
xyPlot(y, x);
```

creates a graph of $y(x)$. It is also possible to plot several curves at once. This is accomplished using the matrix concatenation operator:

```
xyPlot(y1 & y2, x);
```

displays $y_1(x)$ and $y_2(x)$ on the same plot. Alternatively, a matrix, whose columns hold the ordinate arrays, can be passed as the first argument.

For most applications, x is a vector. However, it is sometimes convenient to have a different abscissa vector associated with each ordinate. For example

```
xyPlot(y1 & y2, x1 & x2);
```

displays $y_1(x_1)$ and $y_2(x_2)$ on the same plot. In this case, y and x must have the same dimensions. An example of a case where this capability is useful is for plotting flux surface shapes. Here, we have a series of (R, Z) pairs describing each flux surface and many of these series describing the complex flux surface geometry.

4.3.2 Basic Contour Plots

Contour plots are made using a call to the routine `contourPlot`. This routine has the prototype

```
Void contourPlot(const RealMatrix &z, const RealVector &x,
  const RealVector &y, const RealVector &clevels, option-list
```

where

- `z` is an $M \times N$ matrix holding values of a function $z(x, y)$ on a grid.
- `x` is a vector of length M holding the x-values z is evaluated on.
- `y` is a vector of length N holding the y-values z is evaluated on.
- `clevels` is a vector of length NC holding the desired contour levels.

The *option-list* refers to an optional set of additional arguments that we will consider later. The values in the x and y arrays need not be evenly spaced. However, there is no mechanism for handling missing data in the z matrix.

4.3.3 Basic 3–D Plots

3–D plots are made using a call to the routine `threeDPlot`. This routine has the prototype

```
Void threeDPlot(const RealMatrix &z, const RealVector &x,
  const RealVector &y, const Real &altitude, const Real &azimuth, option-list
```

where

- `z` is an $M \times N$ matrix holding values of a function $z(x, y)$ on a grid.
- `x` is a vector of length M holding the x-values z is evaluated on.
- `y` is a vector of length N holding the y-values z is evaluated on.
- `altitude` is the altitude angle, in degrees, of the point the plot is being viewed from.
- `azimuth` is the azimuth angle, in degrees, of the point the plot is being viewed from.

The *option-list* refers to an optional set of additional arguments that we will consider later. The values in the `x` and `y` arrays need not be evenly spaced. However, there is no mechanism for handling missing data in the `z` matrix.

4.3.4 Options

In addition to simply plotting curves, one usually wants to add, for instance, labels, a title, colors, line styles, marks, etc. This is accomplished using the *option-list* referred to above. An *option-list* is a comma separated list of zero to ten *option-specifications*. An *option-specification*, in turn, has the syntax

option = expression

where an *option* is simply an identifier. Therefore, an example of an X–Y plot call with a non-empty *option-list* is

```
xyPlot(y, x, xLabel = ''x'', yLabel = ''y'', lineColor = red);
```

This gives x-axis and y-axis labels and indicates that the curve should be drawn in red.

The possible options are summarized below.

- `axesColor` (XY, CP, 3D[1]) Sets the color of the axes. Possible choices are: `coral`, `red`, `yellow`, `green`, `aquamarine`, `pink`, `wheat`, `grey`, `brown`, `blue`, `blueViolet`, `cyan`, `turquoise`, `magenta`, `salmon`, and `white`. Default value is `white`.

- `scale` (XY, CP) Sets scaling of the axes. If `scale` is `normal`, the axes are scaled independently to use as much of the screen as possible; if `scale` is `equal` the scales of the x and y axes is made equal. Default value is `normal`.

- `axes` (XY, CP) Controls drawing of the box (axes) around the plot. Possible choices are `noBox` (no axes or annotation), `boxOnly` (draw the bounding box only), `normalAxes` (indicating draw the bounding box with coordinate values around the edge), `axisXOYO` (same as `normalAxis` but also draw the lines $x = 0$ and $y = 0$), `axisXOYOGrid` (same as `axisXOYO` but also draw a grid), `logLin` (logarithmic x axis, linear y axis), `logLinYO` (same as `logLin` but also the line $y = 0$), `linLog` (linear x axis, logarithmic y axis), `linLogXO` (same as `linLog` but also the line $x = 0$), and `logLog` (logarithmic x and y axes). Default value is `normalAxes`.

- `labelColor` (XY, CP, 3D) Sets the color of the axis labels and the title. See the description of `axesColor` for possible choices. Default value is `white`.

- `labelFont` (XY, CP, 3D) Sets the font of the axis labels and the title. Possible choices are `simple` (a sans serif font), `italic`, `roman`, and `script` (a handwriting font). Default value is `simple`.

- `xLabel` (XY, CP, 3D) Sets the text for the x axis label. Any character string can be assigned. It is possible to include escape sequences to trigger greek letters, subscripts, superscripts, etc. This will be discussed later. Default value is the empty string.

- `yLabel` (XY, CP, 3D) Sets the text for the y axis label. Any character string can be assigned. Default value is the empty string.

[1] XY = applicable to X-Y plots, CP = applicable to contour plots, 3D = applicable to 3-D plots.

- zLabel (3D) Sets the text for the z axis label. Any character string can be assigned. Default value is the empty string.
- title (XY, CP, 3D) Sets the text for the plot title. Any character string can be assigned. Default value is the empty string.
- mark (XY) Sets the symbol used to mark data points. Possible choices are noMark, box, plus, asterisk, dot, circle, cross, triangle, crossHair, bullsEye, diamond, star, solidBox, solidCircle, solidStar, or cycle. The cycle option is special: it will be described in more detail later: Default value is noMark.
- markColor (XY) Sets the color of the marks. See the description of axesColor most possible choices. In addition, the user may set mark to cycle. Default value is white.
- lineColor (XY, CP, 3D) Sets the color of the lines. See the description of axesColor most possible choices. In addition, the user may set lineColor to cycle for X–Y plots. Default value is white.
- lineStyle (XY, CP, 3D) Sets the style of the lines. Possible choices are solid, dashedN (where N is 1–7), or cycle. The cycle value applies to X–Y plots only. Default value is solid.
- surface (3D) Sets cross-hatching on 3–D surface. Possible choices include xLines (lines are drawn parallel to x axis), yLines (lines are drawn parallel to y axis), and xyLines (lines are drawn parallel to x axis and y axis). Default value is xyLines.
- meshPlot (3D) Tells whether or not to plot a 3–D surface mesh. Possible choices are yes and no. Default value is no.

One can include escape sequences in the character strings that are assigned to the xLabel, yLabel, zLabel, and title options. All escape sequences start with the number symbol (#). The following escape sequences are possible:

- #u Move up to the superscript position (ended by #d).
- #d Move down to the subscript position (ended by #u).
- #b Backspace (to allow overprinting).

- ## The number symbol.
- #+ Toggle overline mode.
- #- Toggle underline mode.
- #gx Greek letter corresponding to the roman letter x (experiment to figure out the mapping).
- #fn Switch to simple font.
- #fr Switch to roman font.
- #fi Switch to italic font.
- #fs Switch to script font.

For example, "#gk#da#u" will lead to κ_a

The options `mark`, `lineColor`, `markColor`, and `lineStyle` can be assigned the special value `cycle`. This value applies only to X–Y plots and is useful in when multiple curves are being plotted. In the case of the `mark` option, this value indicates that the mark should cycle through the series of values stored in the array `Plot::markCycle`. The first curve will be marked according to `Plot::markCycle(1)` and so on until the end of the array is reached. Then, the marks will restart the cycle at the beginning. The same goes for `lineColor` and `markColor` (which cycle through the `Plot::colorCycle` arra) and `lineStyle` (which cycles through the `Plot::styleCycle` array. The user can adjust how the cycle goes by resizing and adjsting the contents of the clycle arrays.

Examples of plotting routine usage are found in the Shell file "plot.sc."

Bibliography

[1] Bjarne Stroustrup: *The C++ Programming Language*, Addison-Wesley, Reading, Massachusetts, 1986.

[2] Brian W. Kernigan and Dennis M. Richie: *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[3] Samuel P. Harbison and Guy L. Steele: *C, A Reference Manual*, Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[4] Bjarne Stroustrup: *The C++ Programming Language*, Addison-Wesley, Reading, Massachusetts, 2nd Edition, 1991.

[5] Stanley B. Lippman: *The C++ Primer*, Addison-Wesley, Reading, Massachusetts, 2nd Edition, 1991.

[6] Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, Massachusetts, 1990.

[7] Grady Booch: *Object Oriented Design with Applications*, Benjamin/Cummings, Redwood City, California, 1991.

[8] Paul F. Dubois, Lee Busby, Peter Willmann, and Janet Takemoto: *MPPL, A More Productive Programming Language.* Available electronically from any NERSC computer by typing first "`cfs get /basis/mppldoc/mpplrpt`" and then "`allout nip mpplrpt gr. box` *ann* `mppl`" where *ann* is your box number.

[9] M.J. LeBrun, G. Furnish, and T. Richardson: *The PLPLOT Plotting Library Programmers Reference Manual Version 4.0*, University of Texas at Austin Report DOE/ET-53088-538, 1992.

WHY A SYSTEMS CODE IS NECESSARY

- The mission goals and the physics/engineering database constraints form a coupled set of non-linear, simultaneous, inequality relations. They cannot be solved in closed form, nor can accurate scaling laws be formulated to obtain an optimum.

- The computational routines (e.g, transport modules, magnet modules, ...) are the function evaluators for the coupled equation set.

- The number of design variables (e.g, R, A, I, B, ...) and operational variables (n, T, J_ψ, ...) typically exceeds the number of equations, $\Rightarrow$ an optimization problem!

- A range of optimization figures-of-merit are possible, e.g,: (i) minimize cost, (ii) maximize ignition margin, (ii) maximize neutron wall loading, ...etc, subject to a given set of mission objectives and constraints. Moreover, we may require a *bi-modal* optimization, e.g: maximize performance within cost constraints for both inductive-ignited and steady-state operation simultaneously.

- A unique ITER design requires that the 4 design variables: $\{I, A, B_{tf}, \kappa(\delta)\}$, be determined. This is a sufficient, but not unique, set -- all other design parameters $(R, a, B_o, \$, ...etc\}$ can be determined from (or substituted for) these datum parameters. A unique performance for inductive-ignited operation is then determined by selection of $\{profiles, <n>, <T>, ...\}$. A unique performance under steady-state current-driven (and hybrid) operation is determined by selection of $\{<n>, <T>, I(q_\psi \geq 3), P_{aux}, profiles, ...\}$, subject to the constraints of beta, confinement(Q), divertor conditions, neutron wall load, hybrid burn time , ...etc.

- $\Rightarrow$ A systems code is the only way to handle these coupled "systems" self-consistently. It is a fast, invaluable "tool" for overall design optimization, sensitivity studies and performance prediction. Any process or system which is quantifiable can be included at the appropriate level.

Obtaining An Optimum ITER Design Point: There Is No Free Lunch!

MISSION OBJECTIVES

- Ignition
- Inductive burn for 1000s
- $Q_{current\text{-}drive} > 5$
- Neutron wall load = 1 MW m^{-2}, . . . etc.

PHYSICS AND TECHNOLOGY DATABASE CONSTRAINTS

- Confinement
- Troyon beta limits
- Divertor heat flux
- Magnet stress limits
- Radiation damage limits ... etc.

SYSTEMS CODE ANALYSES

Search parameter space within constraint boundaries for minimum cost machine which fulfills mission objectives:

- single point optimization method
- I-A-B_{tf}-κ space method

→ OPTIMUM ITER BASELINE DESIGN

Cheaper, smaller ITER designs are possible only by:

(1) Reducing the mission goals (e.g., Q = 10 instead of ignition, τ_{burn} = 100s instead of 1000s, . . . etc.)

(2) Increasing the risk (e.g., increase magnet stress limits from 600 MPa to 700 MPa, . . . etc.)

(3) Improving the physics database (e.g., confinement and beta enhancements of ~ 1.5 x 2 through current profile control, . . . etc.)

(dj9111174v/ljpv-m5)

Rationale

- Plasma profiles (n,T,J) are of increasing significance in the selection and optimization of design concepts. Once the EDA baseline is defined, profile-consistency will be of paramount importance in investigating its operational performance.

- Increasingly complex questions on ITER operation and design are being addressed to us, which are stretching the present U.S. Systems Codes TETRA and QUICK to the limits of their intended ability.

- In the past, turn-around time for many ITER scoping and operations studies have taken several months. For established techniques, this is too long by a factor of ~10^4! Most ITER analysis tasks which have an established quantitative formalism can be coded and quickly analyzed within a comprehensive systems code.

- We are now approaching the limits of O–D modeling and must acknowledge the consequences of radial dependencies. However, such dependencies (e.g., transport) are subject to uncertainty so we must continue to assess sensitivities through parametric studies. ⇒ Need for <u>fast</u> radially-dependent techniques.

- The EDA design point and its operational performance <u>must</u> be robust relative to uncertainties in profiles.

Prospective Features of SUPERCODE

- Fast, time-dependent 1-1/2 D Systems and Operational Code; lifetime ~5–7 years
- A prospective international code for the ITER EDA, having full international access, participation and control
- Physics routines based on variational solution techniques—a vast reduction in CPU time with accuracies to within a few % of exact formalisms
- Optimization in multi-variable (~30) space subject to database constraints
- Portable to most UNIX platforms: Cray, Workstations, Macintosh
- Macintosh-like graphical user interface – GUI (if it's not easy to use, people won't use it!)
- Modular with an executive driver-shell (if it's not easy to modify, people won't use it!)
- Post processor with interactive, animated, graphics (design decisions require insight not numbers; most codes provide numbers not insight)**
- Prospective users/contributors:

	U.S.	ITER International?
Users	20+	40+?
Contributors	~10	~20?

** There is, at present, a gross imbalance between the time and effort devoted to development of analytic tools and that devoted to formalisms for rational assessment of the results.

Prospective Applications

- Parametric trade studies of new configuration options
- Design optimization
- Analyses of operational modes (inductive-ignition, hybrid, steady-state current-driven, LH-assisted ramp-up, etc.)
- Sensitivity and uncertainty analysis
- Cost-risk-benefit analysis and applications of decision analysis to pinpoint critical design issues and choices
- Burn control
- Emergency shutdown
- Assistance with PF modeling and optimization
- Modeling of start-up/shutdown scenarios
- Optimization of divertor performance
- Investigation of the impact of new divertor and edge physics initiatives (e.g., impurity seeding, gas injection, divertor biasing, etc.)
- Investigation of the impact of new current drive initiatives (e.g., edge helicity injection, etc.)

It is important to appreciate that SUPERCODE will be able to perform such studies with self-consistent recognition of mission and database constraints.

Prospective Uses

CALCULATIONS:

- MHD Equilibrium.
- 0D and 1-1/2D Transport.
- Coupled physics and engineering with costing.

MAJOR OPERATING MODES:

- Steady-state.
  - POPCONs.
  - I-a-B-κ scan.
- Optimization.
  - Minimum cost.
  - Minimum major radius.
- Time-dependent.
  - Burn control.
  - Emergency shut-down.
  - Volt-second consumption and availability.

Why the Systems Code Should Incorporate Profile Effects

Power Balance:	P_{fus}, P_{rad}, ignition, etc., are strong functions of n(r), T(r).
Beta Limits:	$g_{Troyon} = f(q_\varphi(r), l_i(J), \alpha_n(\chi), \alpha_T(\chi))$. New results from TFTR and D-III show strong dependence on J.
Confinement:	New results show strong dependence on J, $\tau_E = f(J)$.
Vertical Stab:	Growth rates, power = $f(l_i(J))$. Rigid and non-rigid analyses predict opposite behavior with $l_i(J)$
Current Drive:	Current drive performance (powers, efficiencies, type (NB, ...), no. ports, ...) is a strong function of profiles; $= f(J, \alpha_n(\chi), \alpha_T(\chi))$
Control:	Burn control is strongly dependent on profiles (pressure(χ), J, profile transients). We have reached limits of O-D modeling.
Divertor:	Divertor conditions are dependent on scrape-off magnetics which are profile-dependent; e.g: $B_{p,plate} = f(\beta_p(\chi), l_i)$.
PF Coil Optimization:	Optimization depends strongly on pressure and current profiles.

⇒ <u>EDA design must be robust relative to uncertainties in profiles. We need fast methods to examine sensitivity to a range of profiles with coupled physics and engineering</u>

PARTIAL LIST OF MISSION OBJECTIVES, DATABASE CONSTRAINTS AND DESIGN VARIABLES

(¶ - possible mission objective for ITER.)

MISSION OBJECTIVE OR CONSTRAINT	TYPICAL CONSTRAINED PARAMETER(S)	TYPICAL ALLOWABLES / REQUIREMENTS	MAJOR CONTROLLING DESIGN VARIABLES[1]
Controlled ignition¶ /power balance[2]	•Confinement time •T stability	≤2.0*ITER-89-P scaling Controllable!	R,A,I,nT,$P_{aux+\alpha}$,profiles[3], fuel, control power, penetration,
Beta	•Troyon coefficient	≤2.5-3.0%[4]	B, I, nT, β_{fast}, profiles[5]
Safety factor	•q_ψ(95%) or q_ψ profile	≥3.0	R, A, I, B, k, δ
Density limit	•Edge density	ITER "Borras" model6	n, P_{aux}, P_{rad}, B
Divertor conditions	•Peak heat flux •Plasma temp. •Collisionality	≤5MW/m^2 ≤30eV ≥1?	n, P_{aux}, P_{rad}, I, B, divertor geometry, S/N -v- D/N
Nuclear testing goals¶	•Neut.wall load, pulse length •Neutron fluence •Testing area/volume	≥1.0MW/m^2, [7]1000s 1-3 MWyr/m^2, [8] ?	n, T, a, (β,...), wall load, N_{cycles}, τ_{burn}(V-sec), availability, T_2 breeding ratio, port space
Pulse length¶	•Inductive burn time •Steady-state	≥200-1000s [9] Q>5	V-sec, I, T, f_{BS}, n, P_{aux}, (H_{div}, β,....)
Vertical stability	•Growth rate •Control power	~≤50Hz? ~10's MW?	k, d, a, Δrwall, J-profile, passive structure, S/N -v- D/N
Current-profile control	•Non-inductive-driven fract. under hybrid operation	≥30%	P_{aux}, CD method[10]
Installed auxiliary power	•=req'd P_{heat} or P_{CD}	~≤150MW	confinement, CD
TF Ripple	•Midplane ripple or profile	≤1.5%/.015% edge/axis	$N_{TF\text{-}coils}$, TF coil radius
Radial and vertical build	•Δr's=1	1	R, a, B, I, Δr_{shield}, $\Delta r_{TF,OH}$, $\Delta r_{div.}$, S/N -v- D/N
Superconducting magnet radiation effects	•Insulator damage •Nuclear heating	≤5.10^9 rads[11] Cyroplant capacity/cost	Δr_{shield} Δr_{shield}
Superconducting magnet design allowables (TF+PF)	•Stress •S/C protection •S/C stability	$\sigma_{VonMises}$≤550MPa ΔT_{dump}≤150K, V≤20kV J/Jcrit≤0.5, ΔT≥2.5K	J, B, r, $\Delta r_{TF,OH}$, τ_{burn}, V-sec, N_{cycles}
Access[12]	•Port size •Maintenance •Flexibility	Port allocation Vert/horiz clearances ?	geometry, PF and TF design, $N_{TF\text{-}coils}$, P_{aux}, S/N -v- D/N
Safety and environment¶	•Afterheat, long-term activity •Tritium inventory	Licensing (site-dependent	Neutron wall load, fluence, materials, T_2 burn fraction
Cost (¶ ??)	•Cost	~≤$5-8B?	Everything!

FOOTNOTES FROM TABLE

[1] All design variables are to some extent dependent on the mission objectives or database constraints. This column indicates only some of the major dependencies

[2] Ignition is a mission goal for ITER . Note that control of ignition will also constrain operating point (n,T, P_{aux},...etc)

[3] Profiles of density, temperature and, very importantly, current for advanced confinement exploration

[4] 2.5% for inductive-ignited operation; 3.0% for current-driven operation where current profile control (>30%) is available

[5] Profiles of density, temperature and, very importantly, current for advanced beta exploration

[6] Separatrix density limit is a function of edge power flux to divertor

[7] Average value. Translates to a peak value of ~1.5MW/m^2 on the outboard midplane at the location of the test modules. This is the value at the test module not at the first wall

[8] Mission goal for CDA was 1MWyr/m^2 with design capable of 3MWyr/m^2. May change in the EDA

[9] Minimum pulse length for physics "equilibrium" (other than current skin times) is ~200s. However, uncertainty in V-sec supply and consumption requires a minimum design requirement of ~1000s. This may be adopted in the EDA

[10] E.g., NB's, LH, ECH, IC, etc

[11] A further safety factor of 3 is used in ITER. N/C magnet solid ceramic insulators may be swelling limited to ~≤4e22n/cm^2 (E_n>0.1MeV); powdered ceramics have higher tolerance

[12] Flexibility of access for testing innovative schemes may be a mission goal for ITER

Modules

Physics

MHD Equilibrium	Transport	Neutral Beams
Bootstrap Current	Divertor	Vertical Stability
PF Coil Currents	V-S Consumption	V-S Availability
ICRH	TAE Modes	Fueling

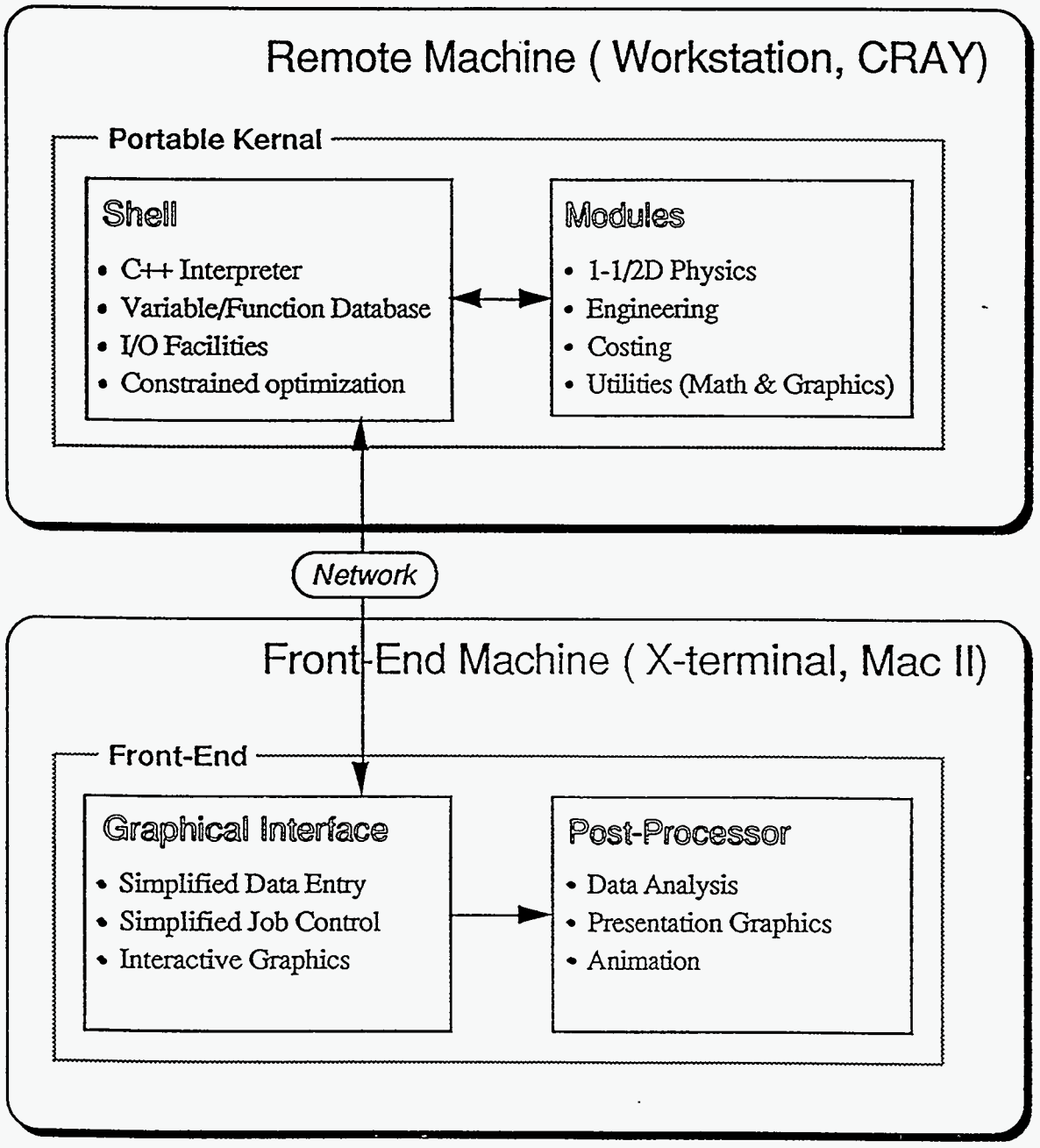
Engineering

AC Power	Access	Buildings
Cost	Cryo Plant	Device Build
Engr. Constraints	Heat Transport	PF Engineering
Injection Power	Power Supplies	Shielding
Structure	Superconductors	TF Engineering
Torus	Tritium System	Vacuum System

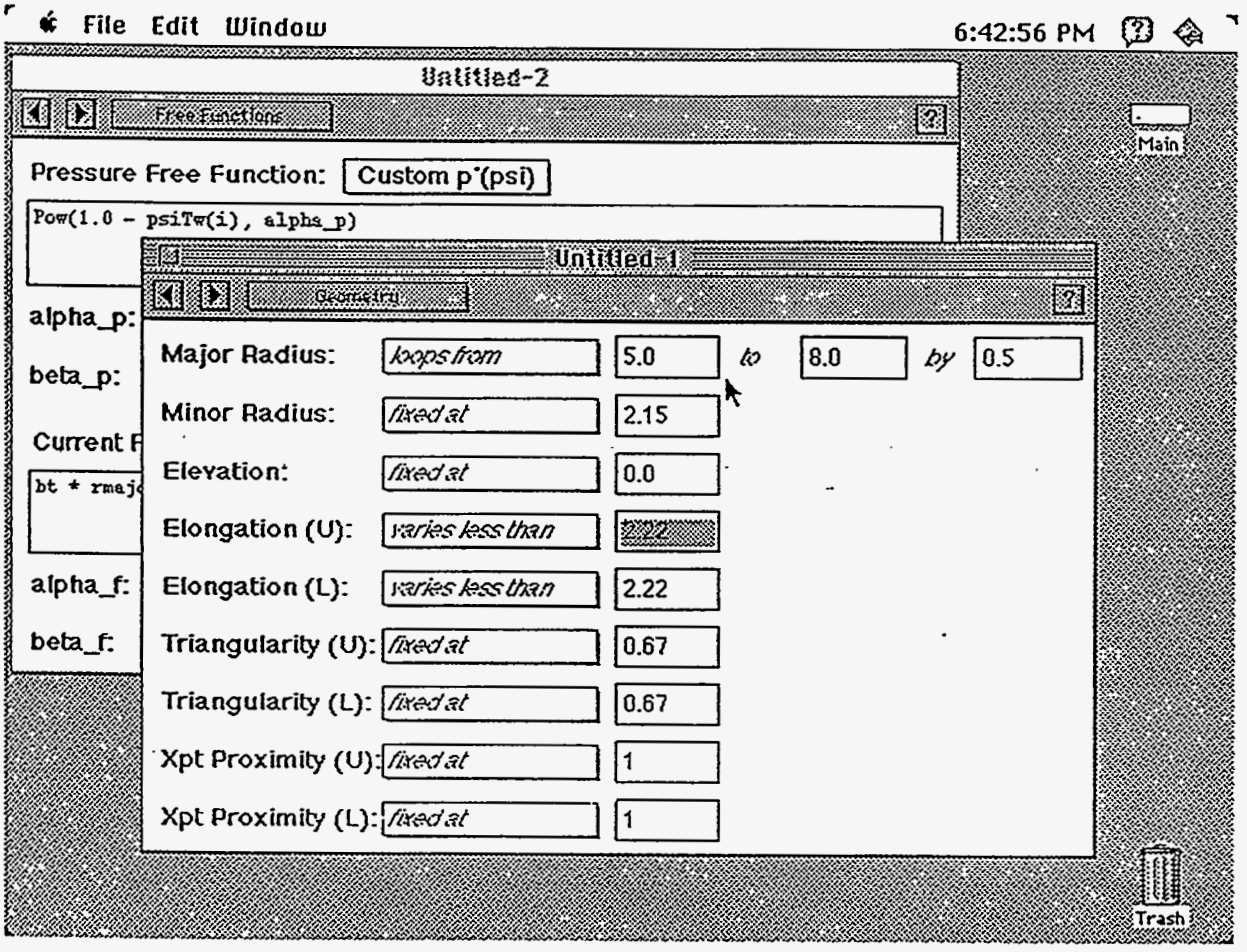
Utilities

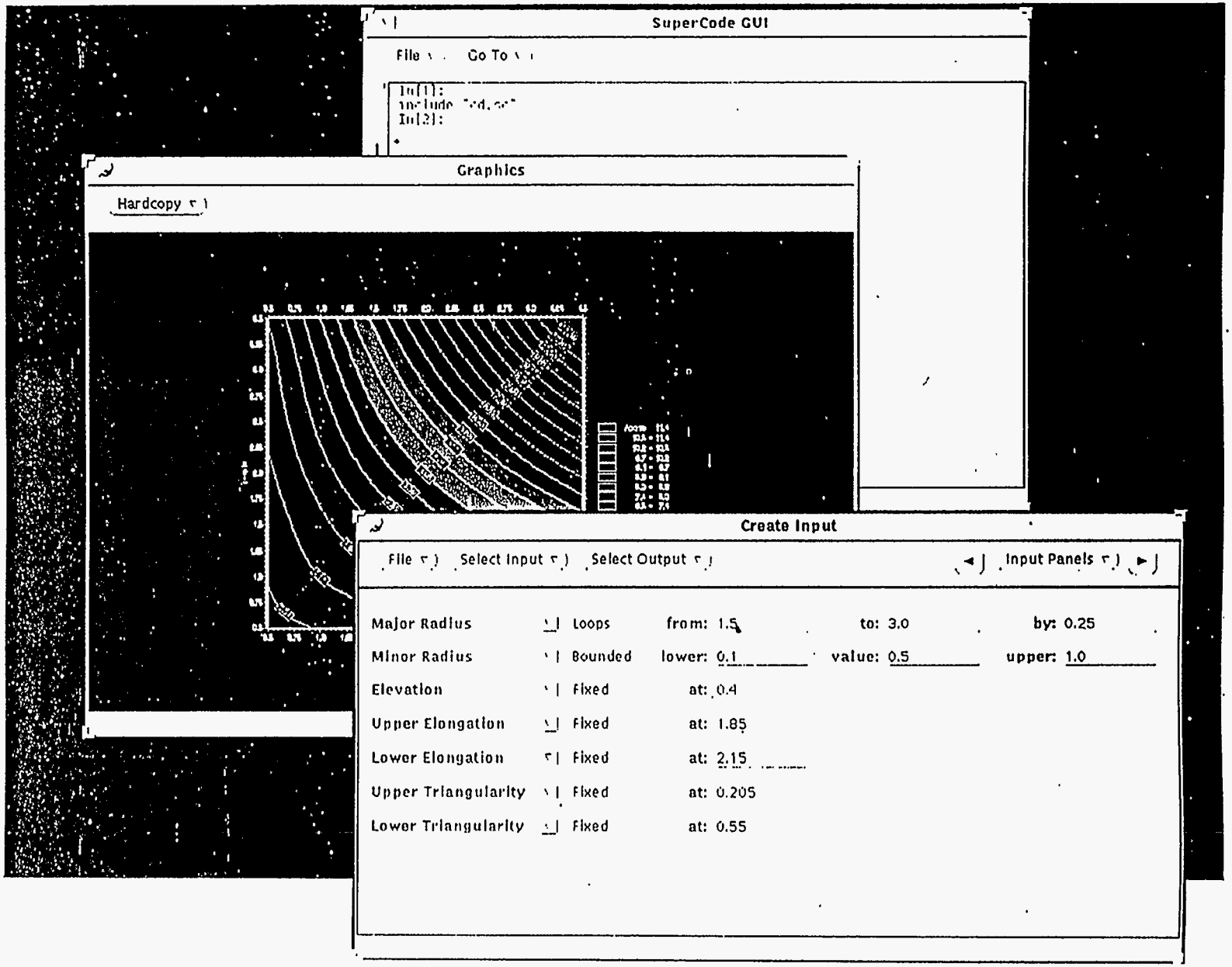
Constants	FFTs	Green's Functs
Linear Algebra	Math Functions	ODE Solution
On-line Help	Optimization	

The ITER SUPERCODE: Architecture

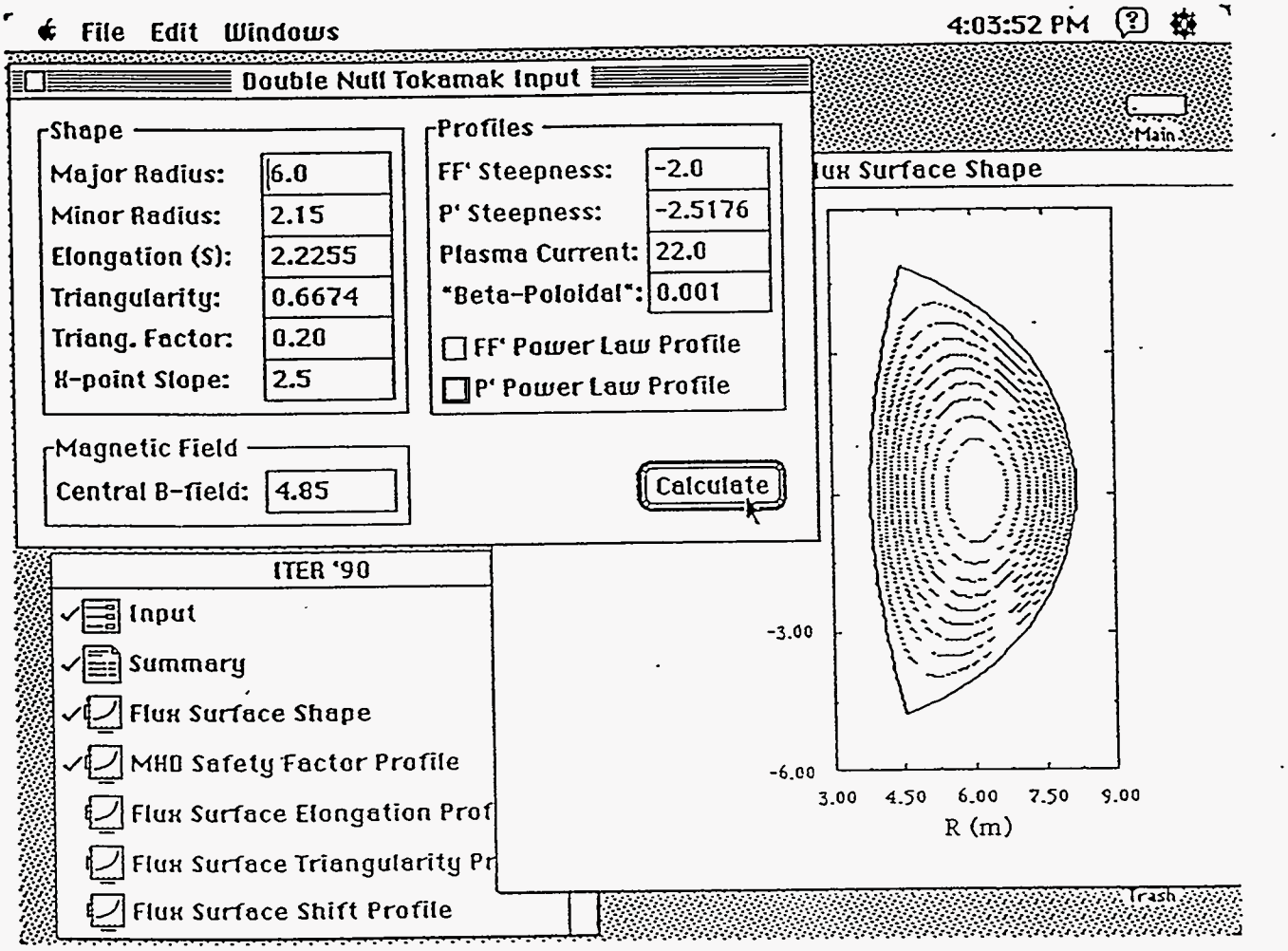


Graphical User Interface




SuperCode GUI
File
Go To
In[1]:
In[2]:
Graphics
Hardcopy
Create Input
File
Select Input
Select Output
Input Panels
Major Radius
Loops
from: 1.5
to: 3.0
by: 0.25
Minor Radius
Bounded
lower: 0.1
value: 0.5
upper: 1.0
Elevation
Fixed
at: 0.4
Upper Elongation
Fixed
at: 1.85
Lower Elongation
Fixed
at: 2.15
Upper Triangularity
Fixed
at: 0.205
Lower Triangularity
Fixed
at: 0.55

Graphical Interface



Variational Equilibrium

GRAD-SHAFRANOV EQUATION:

$$\psi_p R^2 \nabla \cdot \left(\frac{\nabla \tilde{\psi}}{R^2} \right) = -\frac{\mu_0 R^2}{\psi_p} \frac{dp}{d\tilde{\psi}} - \frac{1}{2\psi_p} \frac{dF^2}{d\tilde{\psi}}$$

- Boundary conditions:

$$\nabla \tilde{\psi} = 0|_{axis}, \quad \tilde{\psi} = 1|_{S_p}$$

LAGRANGIAN:

$$\mathcal{L}_{eq}[\psi] = \int_{V_p} \left[\left(\psi_p \frac{\nabla \tilde{\psi}}{R} \right)^2 - 2\mu_0 p(\tilde{\psi}) - \frac{F^2(\tilde{\psi})}{R^2} \right] dV$$

Equilibrium Trial Functions

Flux Function:

$$\tilde{\psi} = \left[\nu\rho^2 + (1-\nu)\rho^4\right]$$

Flux Surface Shapes:

$$\begin{aligned} R &= R_0 + aS(\rho) + a\rho\cos[\mu + D(\rho)\sin\mu] \\ &\quad - a\rho B(\rho)\sin^2\mu + ac_R X(\rho,\,\mu)\rho^4\sin\mu \\ Z &= Z_0 + a\rho E(\rho)\sin\mu + ac_Z X(\rho,\,\mu)\rho^4 \end{aligned}$$

where

$$\begin{aligned} S(\rho) &= \sigma(1-\rho^2) \\ D(\rho) &= \sin^{-1}(\delta - b)\left[\eta\rho + (1-\eta)\rho^3\right] \\ B(\rho) &= b\rho^4 \\ E(\rho) &= \kappa_0 + \lambda\rho^2 + (\kappa - \lambda - \kappa_0)\rho^4 \\ X(\rho,\,\mu) &= \left(\sqrt{1+\alpha} - \sqrt{1-\alpha}\right)\rho\sin\mu \\ &\quad + \sqrt{1-\alpha\rho\sin\mu} - \sqrt{1+\alpha\rho\sin\mu} \end{aligned}$$

and

- ρ and μ are fixed radial and angular coordinates.
- R_0, a, δ, κ, etc. specify the plasma shape.
- $\alpha = \alpha(\mu)$ gives proximity of separatrix surface.
- ν, σ, η, κ_0, and λ are variational parameters.

Variational Solution Procedure

1. Set gradient of $\mathcal{L}_{eq}$ to zero:

   - Yields 7 non-linear equations.

2. Set gradient of $\mathcal{L}_{tr}$ to zero for each of N transport equations:

   - *Time independent.* Yields $4N$ non-linear equations.

   - *Time dependent.* Yields $3N$ ODEs + N non-linear equations.

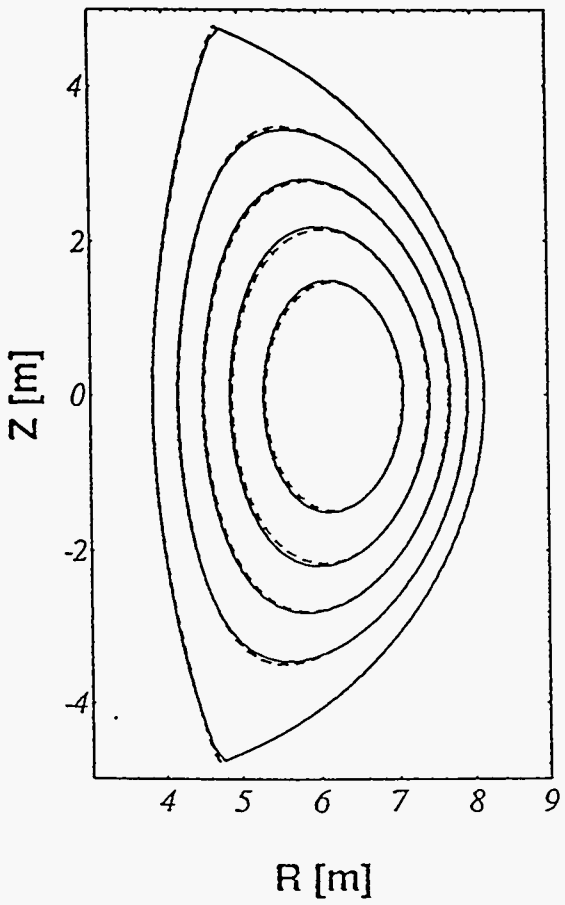
3. Solve simultaneously.

POSSIBLE TO DO 1-1/2-D TRANSPORT CALCULATIONS ON A MACINTOSH II:

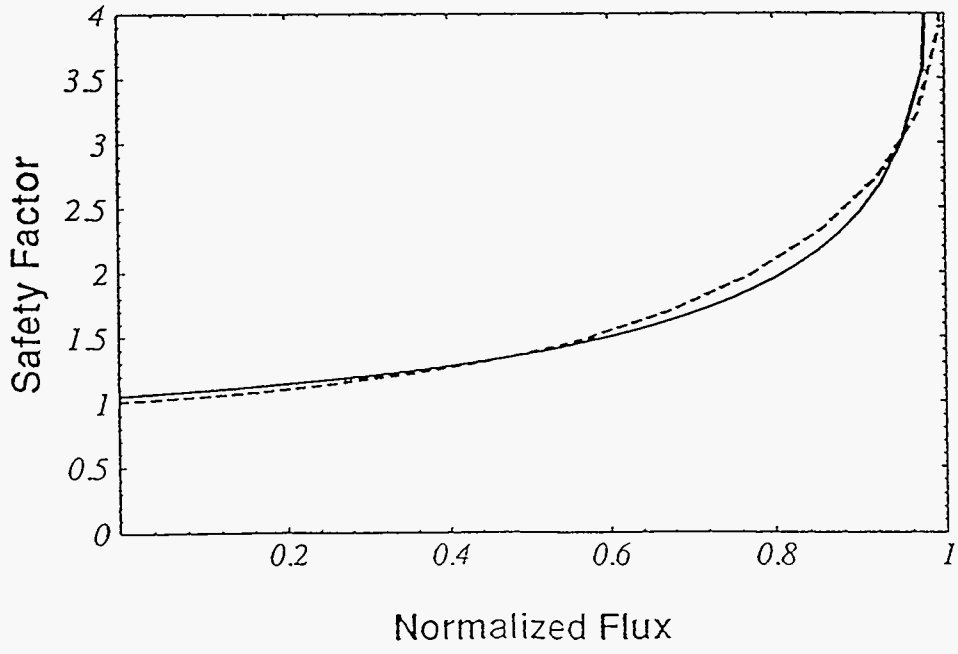
- Time independent cases require 1/2–2 minutes

- Time dependent cases require 5+ minutes

Equilibrium Benchmark



Parameter	TEQ	SUPERCODE
MHD Safety Factor, $q(0)$	1.05	1.01
Toroidal beta, β_t	0.042	0.041
Poloidal beta, β_p	0.63	0.61
Norm. Int. inductance, ℓ_i	0.66	0.64
Axis elongation, κ_0	1.67	1.7
95% elongation, κ_{95}	1.99	2.05
95% triangularity, δ_{95}	0.37	0.44
Flux (axis – edge), ψ_a [Wb]	15.04	15.04
CPU time (s)	10	0.2



Variational Transport

GENERIC TRANSPORT EQUATION:

$$a\frac{\partial U}{\partial t} = \frac{\partial}{\partial \tilde{\psi}}\left(b\frac{\partial U}{\partial \tilde{\psi}}\right) + c$$

- Initial condition: $U(\tilde{\psi}, 0) = U_I(\tilde{\psi})$.
- Boundary conditions:

$$b\frac{\partial U}{\partial \tilde{\psi}}\bigg|_{\tilde{\psi}=0} = 0, \quad AU - \frac{\partial U}{\partial \tilde{\psi}}\bigg|_{\tilde{\psi}=1} = 0$$

LAGRANGIAN:

$$\mathcal{L}_{tr}[U] = \int_0^\tau \int_0^1 \left[\frac{a}{2}\left(U^\dagger\frac{\partial U}{\partial t} - U\frac{\partial U^\dagger}{\partial t}\right) + b\frac{\partial U}{\partial \tilde{\psi}}\frac{\partial U^\dagger}{\partial \tilde{\psi}} - \int_{U_0}^{U} c\,dU' + \int_{U_0}^{U^\dagger} c\,dU' - bAUU^\dagger\delta(\tilde{\psi}-1) - aU_IU^\dagger\delta(t)\right] d\tilde{\psi}\,dt$$

- $U^\dagger$ satisfies

$$a\frac{\partial U^\dagger}{\partial t} = -\frac{\partial}{\partial \tilde{\psi}}\left(b\frac{\partial U^\dagger}{\partial \tilde{\psi}}\right) - c$$

- Initial condition: $U^\dagger(\tilde{\psi}, \tau) = U(\tilde{\psi}, \tau)$.
- Boundary conditions: same as for U.

Transport Trial Functions

DENSITY AND TEMPERATURE:

$$U(\tilde{\psi}, t) = U_0(t)[1 - \lambda_U(t)] \exp[a_U(t)\tilde{\psi} + b_U(t)\tilde{\psi}^2]$$

- U_0, a_U, b_U are variational parameters.
- λ_U is used to satisfy boundary conditions.

FLUX:

$$\Phi(\tilde{\psi}, t) = \Phi_0(t) \int_0^{\tilde{\psi}} [1 + \psi'(\lambda_{\Phi 1}(t) + \lambda_{\Phi 2}(t)\psi')]^{1/2} \exp[\psi'(a_\Phi(t) + b_\Phi(t)\psi')]\langle J/R^2 \rangle \, d\psi'$$

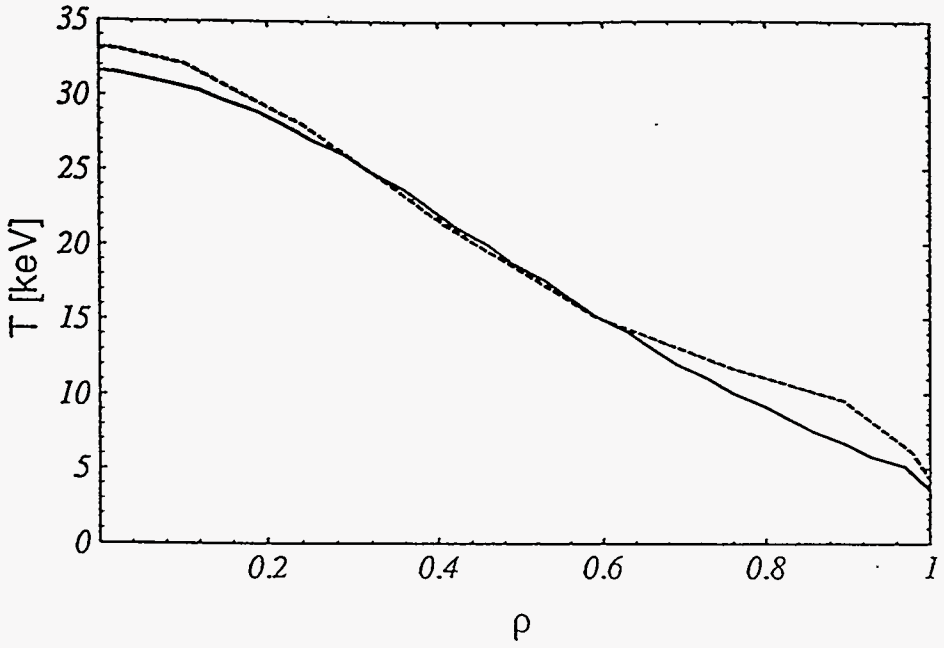
- Φ_0, a_Φ, b_Φ are variational parameters.
- $\lambda_{\Phi 1}$ and $\lambda_{\Phi 2}$ are used to satisfy boundary conditions.

ADJOINT DENSITY, TEMPERATURE AND FLUX:

$$U^\dagger(\tilde{\psi}, t) = c_{U1}(t)U(\tilde{\psi}, t) + c_{U2}(t)\frac{\partial U}{\partial a_U}(\tilde{\psi}, t) + c_{U3}(t)\frac{\partial U}{\partial b_U}(\tilde{\psi}, t)$$

- c_{U1}, c_{U2}, and c_{U3} are variational parameters.

Transport Benchmark



$\chi \sim 1+3\tilde{\psi}$

Parameter	WHIST	SUPERCODE
Density-weighted Average Temp., $\langle T \rangle_n$[keV]	15.9	15.9
Peak Temp., $T(0)$[keV]	31.6	33.2
Avg. Density, $\langle n \rangle$[10^{20}/m^3]	0.995	0.995
Fusion Power, P_{fus}[MW]	1487	1467
Rad. Power, P_{rad}[MW]	46	54
Axis elongation, κ_0	1.65	1.55
MHD Safety Factor, $q(0)$	0.5	0.85
CPU time (s)	12	0.1

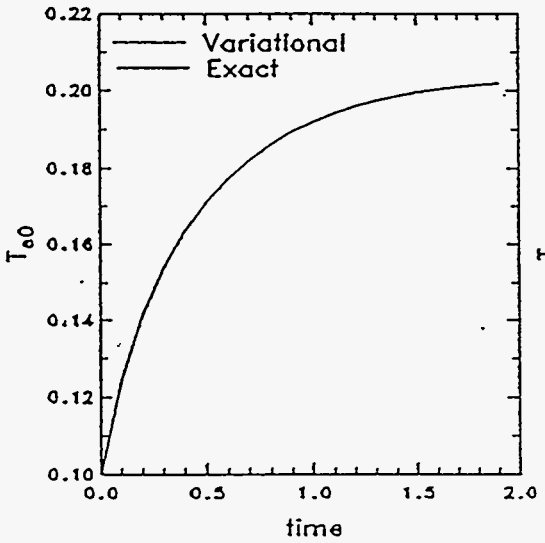
Time-dependent Transport Example

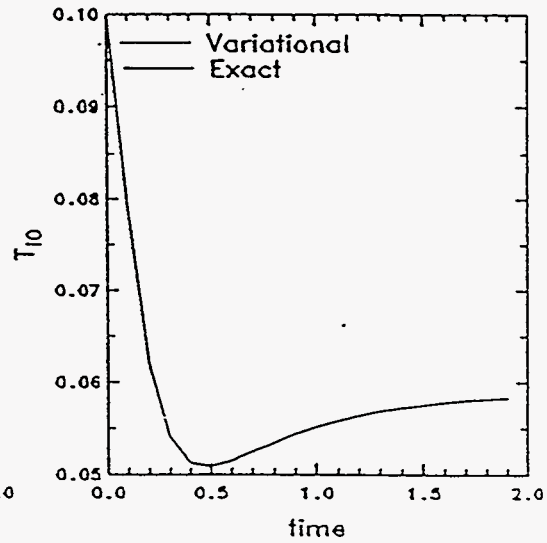
Model Problem:

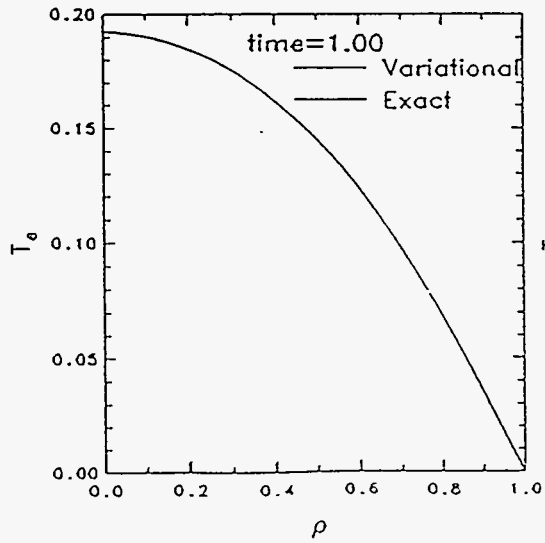
$$\frac{\partial T_e}{\partial t} = \frac{1}{\rho}\frac{\partial}{\partial \rho}\left(0.1\rho \exp(\rho^2)\frac{\partial T}{\partial \rho}\right) + \frac{1}{(T_e+2)^{1.5}} + 5\frac{T_i - T_e}{(T_e+2)^{1.5}}$$

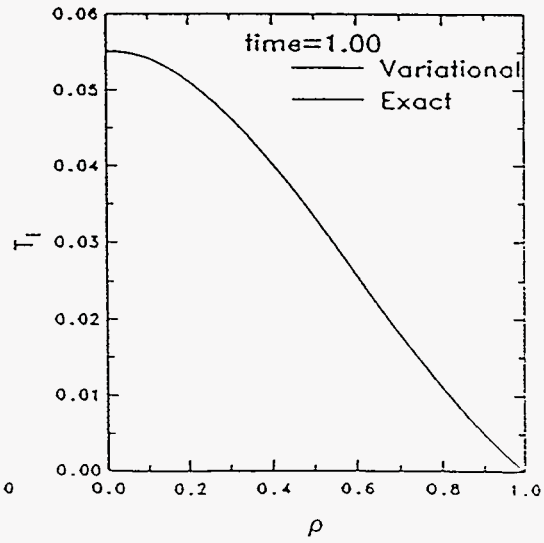
$$\frac{\partial T_i}{\partial t} = \frac{1}{\rho}\frac{\partial}{\partial \rho}\left(0.5\rho \exp(\rho^2)\frac{\partial T}{\partial \rho}\right) - 5\frac{T_i - T_e}{(T_e+2)^{1.5}}$$

$$T_e(1,t) = T_i(1,t) = 0$$



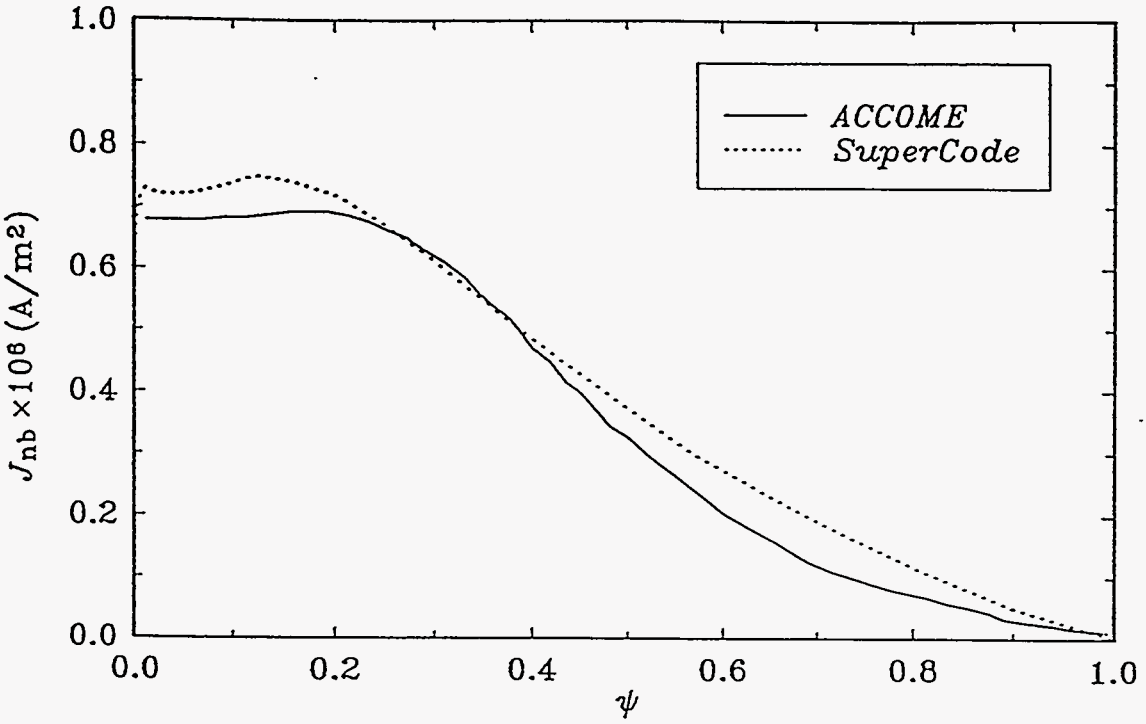






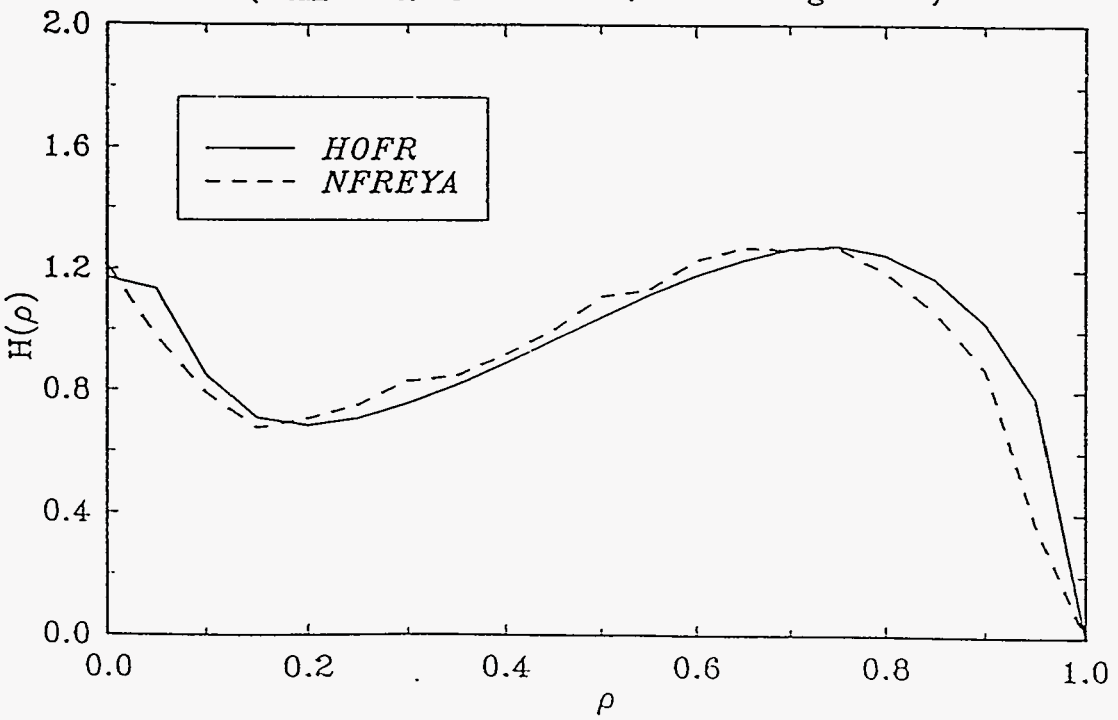
Neutral Beam Benchmark

NB MODULE FOR THE SUPERCODE
(Comparison with the ACCOME code for the B6 SS case)



HOFR and NFREYA comparison

($R_{tan} = R_0$, $E_b = 1.3$ MeV, ITER CDA Ignition)



CPU time difference is $\gtrsim 100$ for SuperCode!

Shell Overview

The shell controls all calculations and I/O for the systems code.

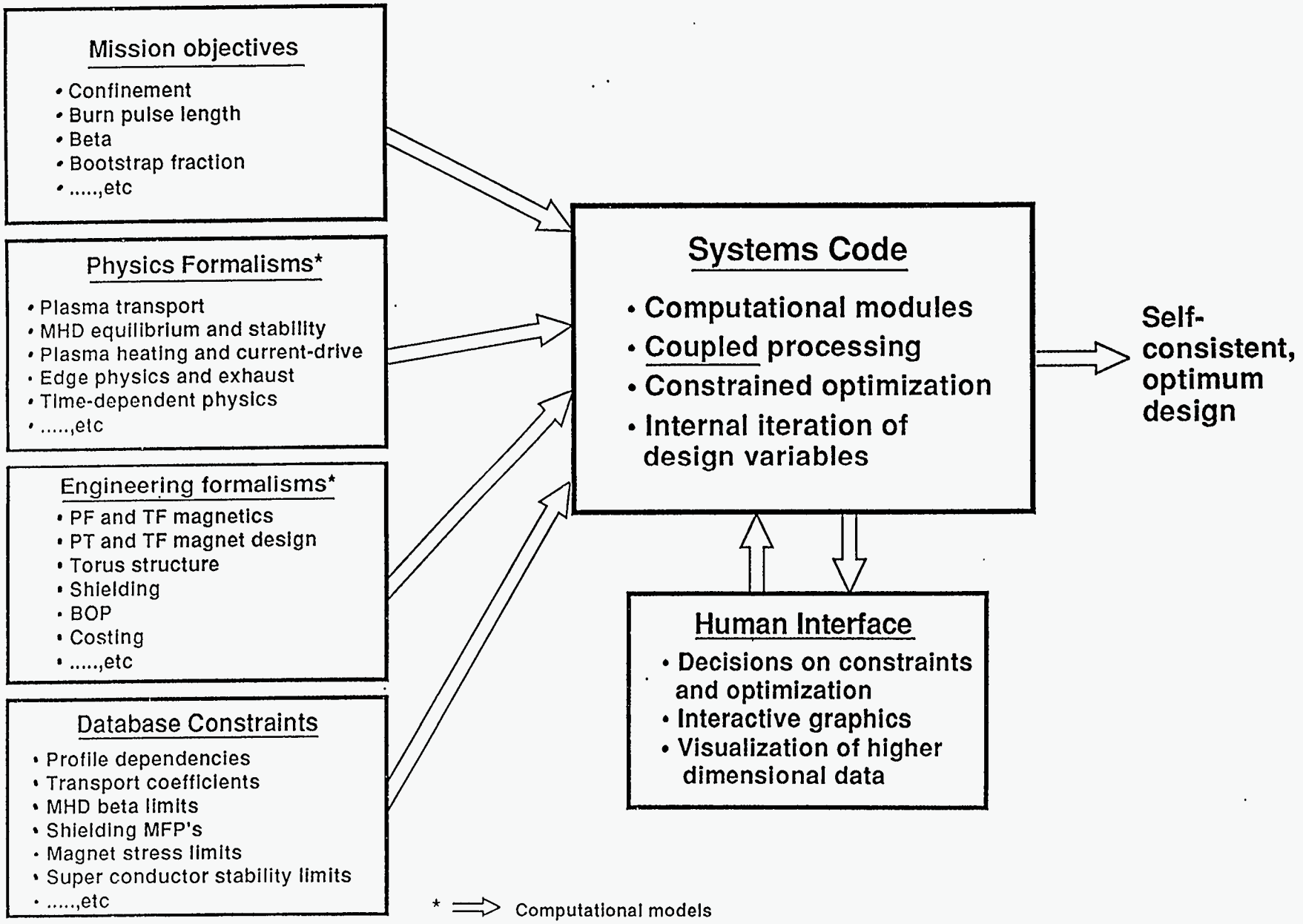
FEATURES:

- C++ syntax.
- Interactive or batch.
- Scalar and matrix arithmetic.
- Decision structures.
- Looping structures.
- Subroutines and functions.
- I/O facilities.
- Compiled variables and functions in modules can be accessed from the shell.
- Interpreted variables and functions can be passed to compiled functions.

Shell Example

```
////////////////////////////////////////////////////////////
//
// SuperCode example: Computes MHD equilibria using ITER
//                    CDA parameters.  Constrains beta to a
//                    desired value.
//
////////////////////////////////////////////////////////////

// Select the ITER CDA parameters

rmajor     = 6.0;    rminor     = 2.15;
kappup     = 2.23;   kappup     = 2.23;
deltup     = 0.6;    deltdn     = 0.6;
asepup     = 1.0;    asepdn     = 1.0;
beansh     = 0.17;   rhoMax     = 0.999;
xptsup     = 2.5;    xptsdn     = 2.5;
alpha_h(1) = -2.5;   alpha_f(1) = -7.5;

// Initialize the basic equilibrium calculation

initEquil();

// We would like to find equilibria with various toroidal
// beta values: add an interpreted constaint to vary the
// pressure on axis to accomplish this

Real betaDesired, betaDiff;
Integer betaCon, betaVar, betaCalc, i = 1;

Void betaDiffCalc()
{
   betaDiff = 1.0 - betat / betaDesired;
}

betaVar = addVariable("pAxis", pAxis, 1.0,
  !Bounded, !Bounded, 0.0, 0.0, On);
betaCon = addConstraint("fix beta-toroidal", betaDiff,
   1.0, Equality, On);
betaCalc = addCalculator("betaDiffCalc()", betaDiffCalc, On);

// Do a scan of the Shafranov shift versus beta

RealMatrix shifts(10);

for (betaDesired = 0.005; betaDesired <= 0.05; betaDesired += 0.005)
{
   doEquil();                        // Compute the equilibrium
   shifts(i++) = r(1,1) - rmajor; // Store the shift
}
```

TPX Design: A systems code is the only way to handle these coupled formalisms self-consistently



20-A-0292-0024B

PARTIAL LIST OF MISSION OBJECTIVES, DATABASE CONSTRAINTS AND DESIGN VARIABLES

(§ - mission objective for ITER. ¶ - mission objective for SSAT)

MISSION OBJECTIVE OR CONSTRAINT	TYPICAL CONSTRAINED PARAMETER(S)	TYPICAL ALLOWABLES FOR *ITER*	TYPICAL ALLOWABLES FOR *TPX*	MAJOR CONTROLLING DESIGN VARIABLES[1]
Ignition§ /power balance[2]	•Confinement time	≤1.8*ITER-89-P	≤2.0-4.0*ITER-89-P ?	R,A,I,nT,$P_{aux+\alpha}$,profiles[3], fuel
Beta¶,[4]	•Troyon coefficient	≤2.5-3.0%[5]	≤3-6%; 1st, 2nd stab.	B, I, nT, β_{fast}, profiles[6]
Safety factor	•q_{ψ}(95%)	≥3.0	≥3.0?	R, A, I, B, κ, δ
Density limit	•Edge density	ITER "Borras" model[7]	Greenwald +Borras?	n, P_{aux}, P_{rad}, B
Divertor conditions¶,[8]	•Peak heat flux •Plasma temp. •Collisionality	≤5MW/m^2 ≤30eV ≥1?	? ? ?	n, P_{aux}, P_{rad}, I, B, divertor geometry
Nuclear testing goals§	•Neutron wall load •Neutron fluence	≥1.0MW/m^2 (average[9]) 1-3 MWyr/m^2 [10]	N.A N.A	n, T, a, (β,...), wall load, τ_{bum}, N_{cycles}
Pulse length§,¶	•Inductive burn time •Steady-state	≥200-1000s [11] Q>5	≥20s	V-sec, I, T f_{BS}, n, P_{aux}, (H_{div}, β,....)
Vertical stability	•Growth rate •Control power	~≤50Hz? ~10's MW?	? ?	κ, δ, a, Δr_{wall}, J-profile
Bootstrap current ¶	•Bootstrap current fraction	Not a mission objective	≥0.66%; ν_{fast}≤0.3	A, β_p(nT, I) , q, profiles
Current-profile control§,¶	•Non-inductive-driven fract.	≥30%	Controlled fraction?	P_{aux}, CD method[12]
Tail electron slow. down	•$\tau_{s(e/e+e/i)}$	≤τ_E/10	ν_{fast}*τ_E≥8.0	E_e, T, I
Installed auxiliary power	•=req'd P_{heat}. or P_{CD}	~≤150MW	≤PPPL site credit?	confinement, CD
Ripple	•Midplane ripple (profile?)	≤1.5%/.015% edge/axis	?	$N_{TF\text{-}coils}$, TF coil radius
Radial and vertical build	•Δr's=1	1	1	R, a, Δr_{shield}, $\Delta r_{TF,OH}$, $\Delta r_{div.}$
Magnet radiation effects	•Insulator damage •Nuclear heating •Activation	≤5.10^9 rads[13] Cyroplant capacity/cost N.A	N.A N.A Hands on access?	Δr_{shield} Δr_{shield} H or D fuel
Magnet design allowables (TF+PF)	•Stress •S/C protection •S/C stability •N/C heat removal	$\sigma_{VonMises}$≤550MPa ΔT_{dump}≤150K, V≤20kV J/J_{crit}≤0.5, ΔT≥2.5K N.A	$\sigma_{VonMises}$≤550MPa? ΔT_{dump}≤150K, V≤20kV? J/J_{crit}≤0.5, ΔT≥2.5K? N.A	J, B, r, Δr_{TF}, Δr_{OH}, τ_{burn}, V-sec, N_{cycles}
Access¶,[14]	•Port size •Maintenance •Flexibility[14]	Port allocation Vert/horiz clearances ?	? ? ?	geometry, PF and TF design, $N_{TF\text{-}coils}$, P_{aux},
Installed power supplies	•=required power supply	N.A	≤PPPL site credit?	P_{aux}, $P_{N/C\ mag}$, $P_{cryo.}$
Cost	•Cost	~≤$5-8B	~≤$500M	everything!

FOOTNOTES FROM TABLE

[1] All design variables are to some extent dependent on the mission objectives or database constraints. This column indicates only some of the major dependencies

[2] Ignition is a mission goal for ITER whereas power balance is a constraint for SSAT. Note that control of ignition will also constrain operating point (n,T, P_{aux},...etc)

[3] Profiles of density, temperature and, very importantly, current for advanced confinement exploration

[4] Beta limits are a constraint for both ITER and SSAT. However, a mission goal for SSAT may be to explore advanced beta regimes

[5] 2.5% for inductive-ignited operation; 3.0% for current-driven operation where current profile control (>30%) is available

[6] Profiles of density, temperature and, very importantly, current for advanced beta exploration

[7] Separatrix density limit is a function of edge power flux to divertor

[8] Divertor conditions are a constraint for ITER but may be a mission goal for SSAT

[9] Translates to a peak value of ~1.5MW/m^2 on the outboard midplane at the location of the test modules

[10] Mission goal is 1MWyr/m^2, design must be capable of 3MWyr/m^2. May change in the EDA to lower fluence goals

[11] Minimum pulse length for physics "equilibrium" (other than current skin times) is ~200s. However, uncertainty in V-sec supply and consumption requires a minimum design requirement of ~1000s. This may be adopted in the EDA

[12] E.g., NB's, LH, ECH, IC, etc

[13] A further safety factor of 3 is used in ITER. N/C magnet solid ceramic insulators may be swelling limited to ~≤4e22n/cm^2 (E_n>0.1MeV); powdered ceramics have higher tolerance

[14] Flexibility of access for testing innovative schemes may be a mission goal for SSAT (it should be for ITER also!)

WHY A SYSTEMS AND OPERATIONAL CODE IS INVALUABLE FOR TPX DESIGN AND OPERATION

- The SSAT mission goals and the physics/engineering database constraints form a coupled set of non-linear, simultaneous, inequality relations. They cannot be solved in closed form, nor can accurate scaling laws be formulated to obtain an optimum.
- The computational routines (e.g, transport modules, magnet modules, ...) are the function evaluators for the coupled equation set.
- The number of design variables (e.g, R, A, I, B, ...) and operational variables (n, T, J_ψ, ...) typically exceeds the number of equations, $\Rightarrow$ an optimization problem!
- A range of optimization figures-of-merit are possible, e.g,: (i) minimize size/cost for a given set of mission objectives, (ii) maximize performance under a maximum cost constraint, ..., etc. Moreover, we may require *multi-modal* optimizations, e.g: minimize size/cost for several distinct operating modes simultaneously (e.g: high 1st stability beta; high BS; advanced confinement; 2nd stability;..., etc)
- A unique SSAT design requires that the 5 design variables: $\{I, A, B_{tf}, \kappa\ (\delta), q_\psi\}$, and the following operational variables: $\{n, T, P_{aux}, J_\psi, P_\psi, fuel\}$, be determined. This is a sufficient, but not unique, set. All other major parameters $(R, a, B_o, \$, I_{BS}, gTroyon, H^*\tau E, f_{ne}, Q_{div}, \tau_{burn}...)$ can be determined from (or substituted for) these.
- $\Rightarrow$ A systems and operational code is the only way to handle these coupled "systems" self-consistently. It is a fast, invaluable "tool" for overall design optimization, sensitivity studies and performance prediction. Any process or system which is quantifiable can be included at the appropriate level.

APPLICATION OF SUPERCODE TO SSAT

(1) Design optimization

Obtain minimum cost machine(s) with multi-modal mission, with coupled physics, engineering and costing.

(2) Cost-v-performance sensitivities

Perform cost-v-performance sensitivity studies around the design point → justify the design point.

(3) Operational flexibility

Assess operational flexibility of final design point via the 1½D capability of the code. (Must be robust relative to uncertainties)

(4) Design option trade offs

Self-consistently compare and contrast design options (SSAT-S, -R, ...) under the same physics, engineering and cost assumptions.

Rationale

- Plasma profiles (n,T,J) are of increasing significance in the selection and optimization of design concepts. Once the EDA baseline is defined, profile-consistency will be of paramount importance in investigating its operational performance.

- Increasingly complex questions on ITER operation and design are being addressed to us, which are stretching the present U.S. Systems Codes TETRA and QUICK to the limits of their intended ability.

- In the past, turn-around time for many ITER scoping and operations studies have taken several months. For established techniques, this is too long by a factor of ~10^4! Most ITER analysis tasks which have an established quantitative formalism can be coded and quickly analyzed within a comprehensive systems code.

- We are now approaching the limits of O-D modeling and must acknowledge the consequences of radial dependencies. However, such dependencies (e.g., transport) are subject to uncertainty so we must continue to assess sensitivities through parametric studies. ⇒ Need for <u>fast</u> radially-dependent techniques.

- The EDA design point and its operational performance <u>must</u> be robust relative to uncertainties in profiles.

Prospective Features of SUPERCODE

- Fast, time-dependent 1-1/2 D Systems and Operational Code; lifetime ~5–7 years
- A prospective international code for the ITER EDA, having full international access, participation and control
- Physics routines based on variational solution techniques—a vast reduction in CPU time with accuracies to within a few % of exact formalisms
- Optimization in multi-variable (~30) space subject to database constraints
- Portable to most UNIX platforms: Cray, Workstations, Macintosh
- Macintosh-like graphical user interface – GUI (if it's not easy to use, people won't use it!)
- Modular with an executive driver-shell (if it's not easy to modify, people won't use it!)
- Post processor with interactive, animated, graphics (design decisions require insight not numbers; most codes provide numbers not insight)**
- Prospective users/contributors:

	U.S.	ITER International?
Users	20+	40+?
Contributors	~10	~20?

** There is, at present, a gross imbalance between the time and effort devoted to development of analytic tools and that devoted to formalisms for rational assessment of the results.

Prospective Applications

- Parametric trade studies of new configuration options
- Design optimization
- Analyses of operational modes (inductive-ignition, hybrid, steady-state current-driven, LH-assisted ramp-up, etc.)
- Sensitivity and uncertainty analysis
- Cost-risk-benefit analysis and applications of decision analysis to pinpoint critical design issues and choices
- Burn control
- Emergency shutdown
- Assistance with PF modeling and optimization
- Modeling of start-up/shutdown scenarios
- Optimization of divertor performance
- Investigation of the impact of new divertor and edge physics initiatives (e.g., impurity seeding, gas injection, divertor biasing, etc.)
- Investigation of the impact of new current drive initiatives (e.g., edge helicity injection, etc.)

It is important to appreciate that SUPERCODE will be able to perform such studies with self-consistent recognition of mission and database constraints.

Prospective Uses

CALCULATIONS:

- MHD Equilibrium.
- 0D and 1-1/2D Transport.
- Coupled physics and engineering with costing.

MAJOR OPERATING MODES:

- Steady-state.
  - POPCONs.
  - I-a-B-κ scan.
- Optimization.
  - Minimum cost.
  - Minimum major radius.
- Time-dependent.
  - Burn control.
  - Emergency shut-down.
  - Volt-second consumption and availability.

Why the Systems Code Should Incorporate Profile Effects

Power Balance:	P_{fus}, P_{rad}, ignition, etc., are strong functions of n(r), T(r).
Beta Limits:	$g_{Troyon} = f(q\varphi(r), l_i(J), \alpha_n(\chi), \alpha_T(\chi))$. New results from TFTR and D-III show strong dependence on J.
Confinement:	New results show strong dependence on J, $\tau_E = f(J)$.
Vertical Stab:	Growth rates, power = $f(l_i(J))$. Rigid and non-rigid analyses predict opposite behavior with $l_i(J)$
Current Drive:	Current drive performance (powers, efficiencies, type (NB, ...), no. ports, ...) is a strong function of profiles; $= f(J, \alpha_n(\chi), \alpha_T(\chi))$
Control:	Burn control is strongly dependent on profiles (pressure(χ), J, profile transients). We have reached limits of O-D modeling.
Divertor:	Divertor conditions are dependent on scrape-off magnetics which are profile-dependent; e.g: $B_{p,plate} = f(\beta_p(\chi), l_i)$.
PF Coil Optimization:	Optimization depends strongly on pressure and current profiles.

⇒ <u>EDA design must be robust relative to uncertainties in profiles. We need fast methods to examine sensitivity to a range of profiles with coupled physics and engineering</u>

Modules

Physics

MHD Equilibrium	Transport	Neutral Beams
Bootstrap Current	Divertor	Vertical Stability
PF Coil Currents	V-S Consumption	V-S Availability
ICRH	TAE Modes	Fueling

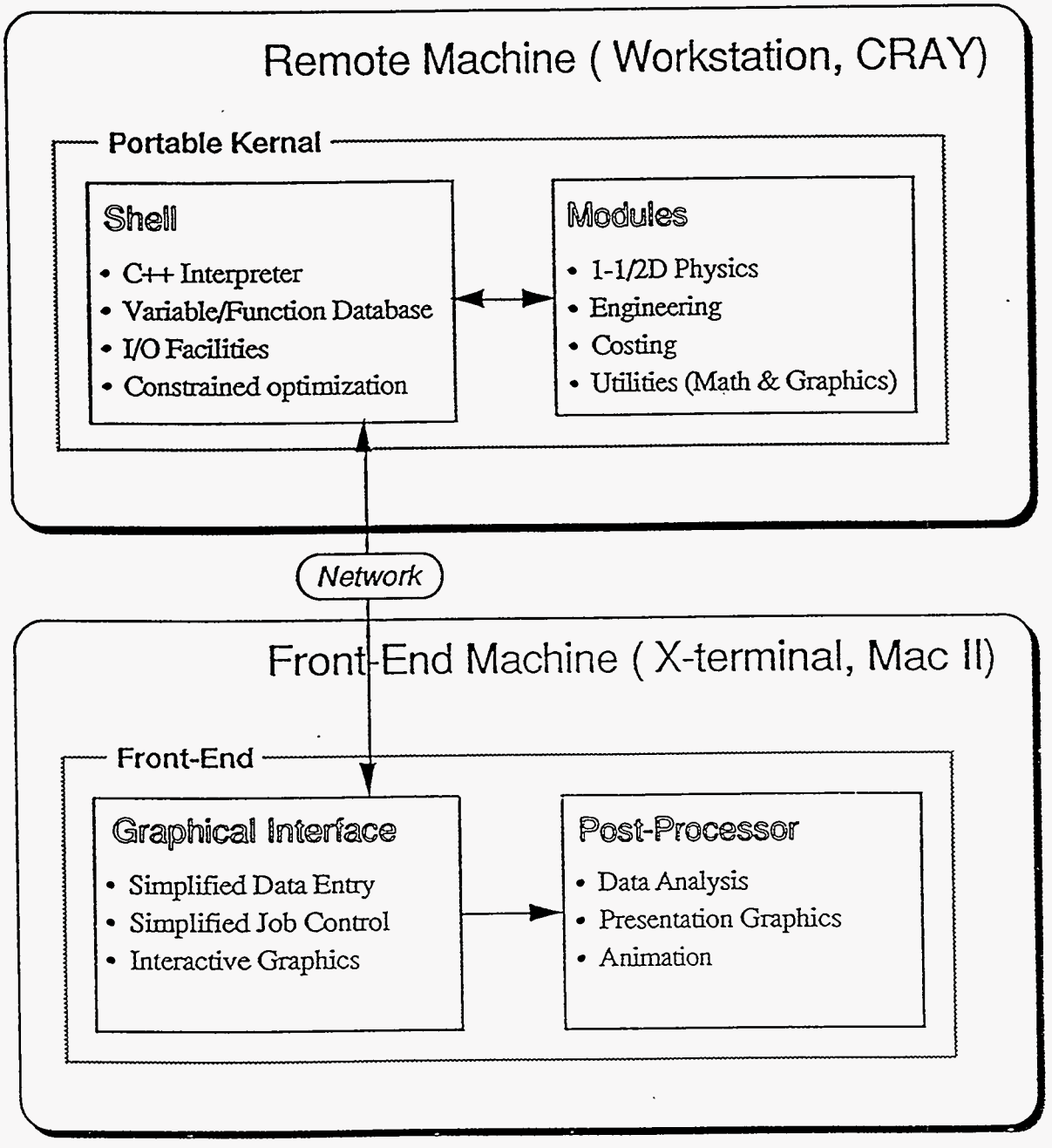
Engineering

AC Power	Access	Buildings
Cost	Cryo Plant	Device Build
Engr. Constraints	Heat Transport	PF Engineering
Injection Power	Power Supplies	Shielding
Structure	Superconductors	TF Engineering
Torus	Tritium System	Vacuum System

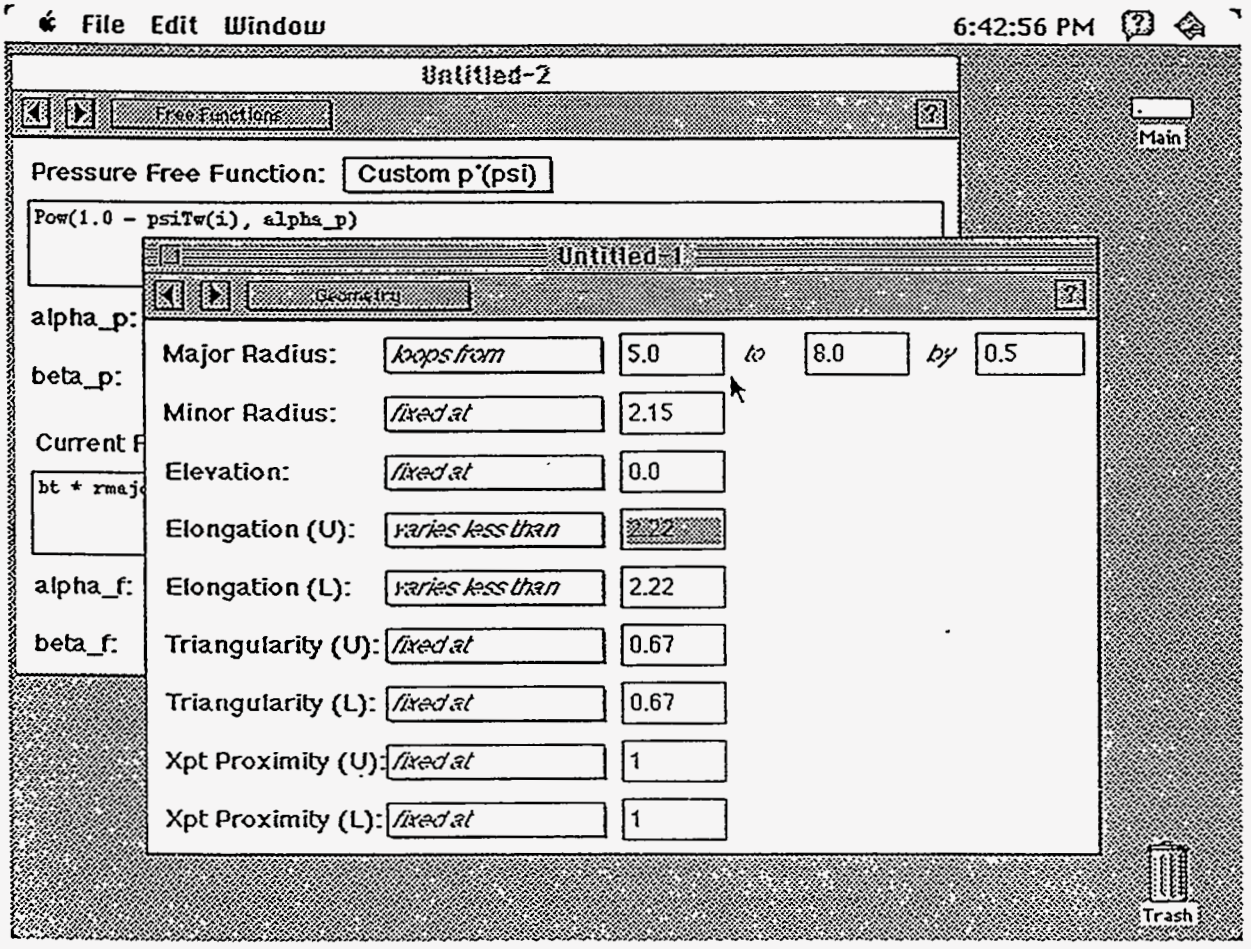
Utilities

Constants	FFTs	Green's Functs
Linear Algebra	Math Functions	ODE Solution
On-line Help	Optimization	

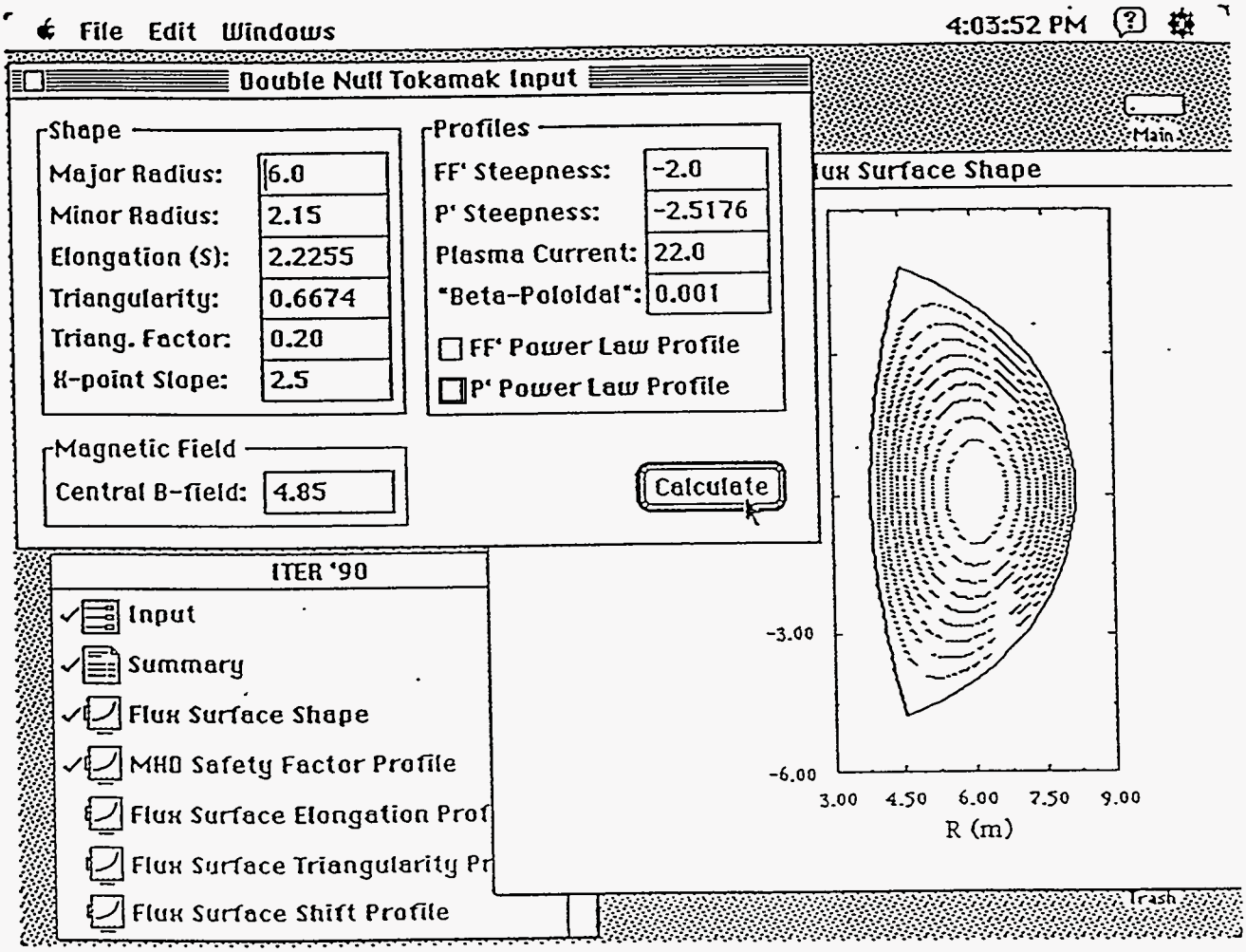
The ITER SUPERCODE: Architecture



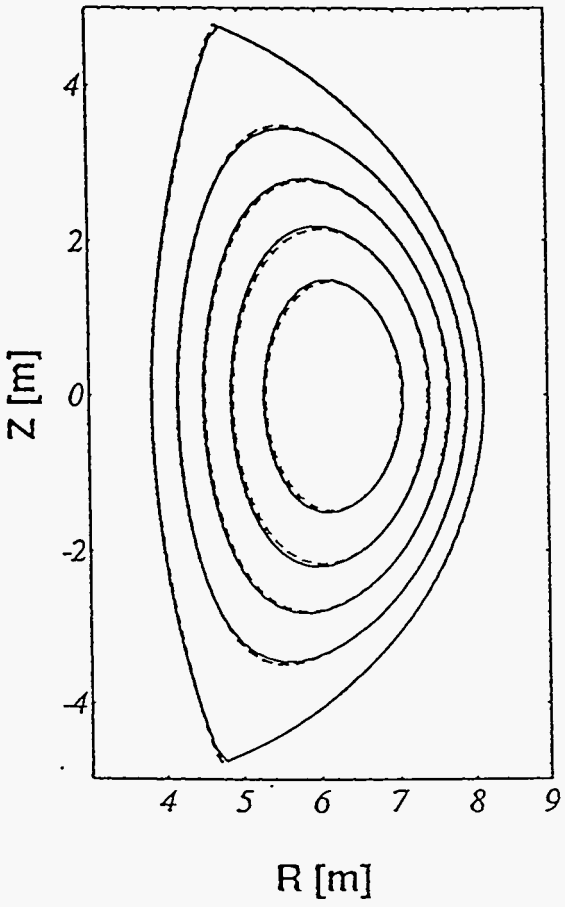
Graphical User Interface



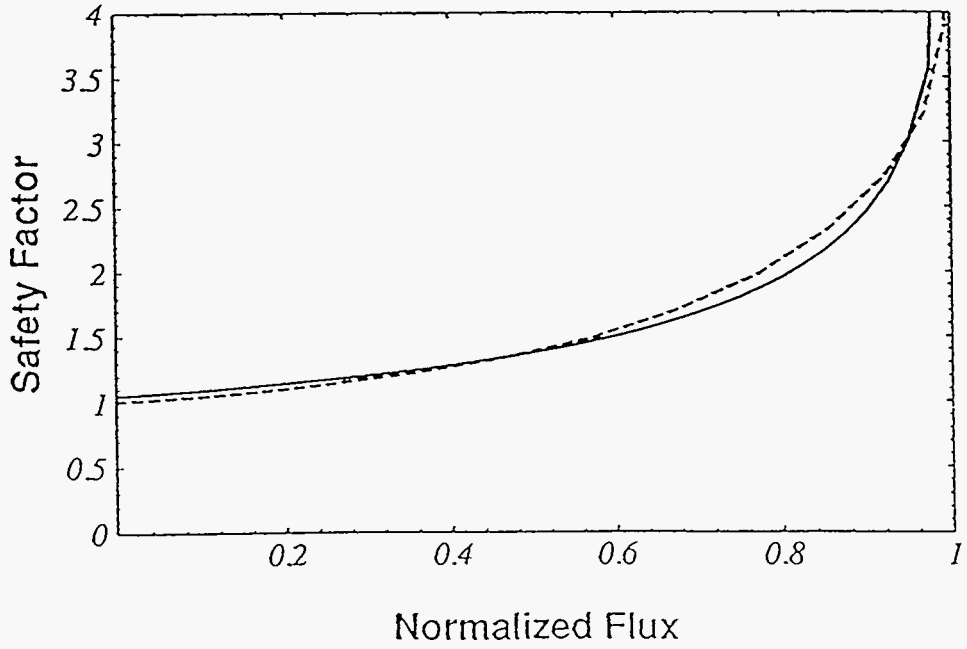
Graphical Interface



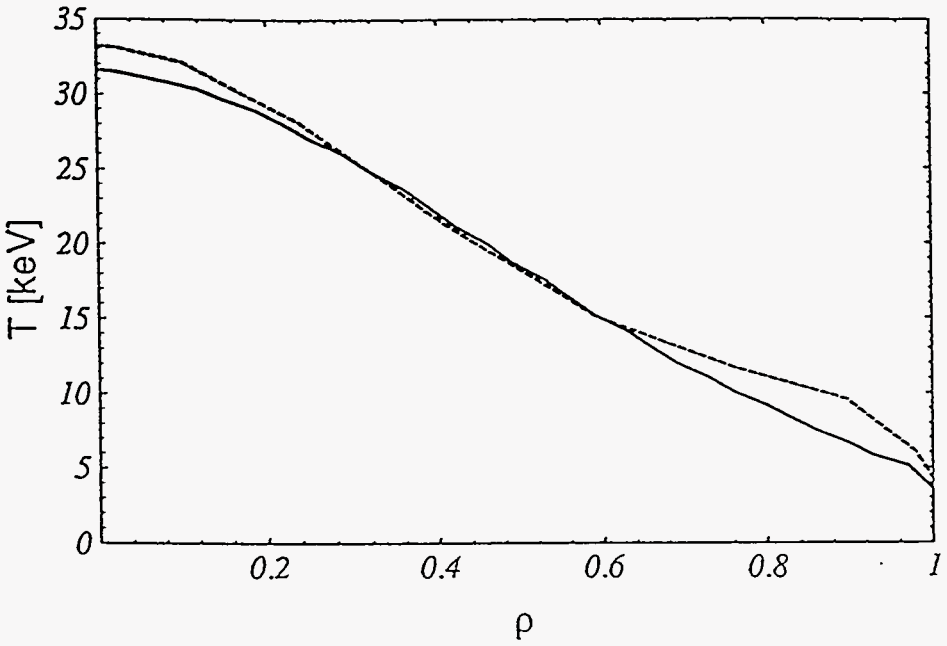
Equilibrium Benchmark



Parameter	TEQ	SUPERCODE
MHD Safety Factor, $q(0)$	1.05	1.01
Toroidal beta, β_t	0.042	0.041
Poloidal beta, β_p	0.63	0.61
Norm. Int. inductance, ℓ_i	0.66	0.64
Axis elongation, κ_0	1.67	1.7
95% elongation, κ_{95}	1.99	2.05
95% triangularity, δ_{95}	0.37	0.44
Flux (axis – edge), ψ_a [Wb]	15.04	15.04



Transport Benchmark

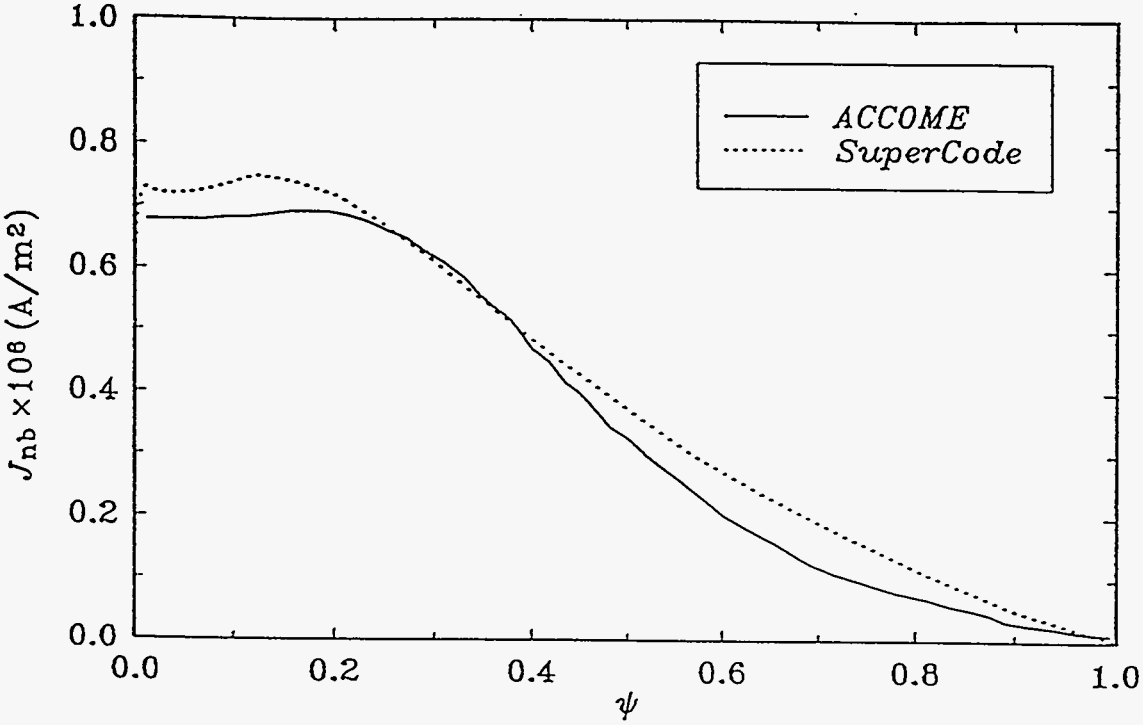


$\chi \sim 1+3\tilde{\psi}$

Parameter	WHIST	SUPERCODE
Density-weighted Average Temp., $\langle T \rangle_n$[keV]	15.9	15.9
Peak Temp., $T(0)$[keV]	31.6	33.2
Avg. Density, $\langle n \rangle [10^{20}/\mathrm{m}^3]$	0.995	0.995
Fusion Power, P_{fus}[MW]	1487	1467
Rad. Power, P_{rad}[MW]	46	54
Axis elongation, κ_0	1.65	1.55
MHD Safety Factor, $q(0)$	0.5	0.85

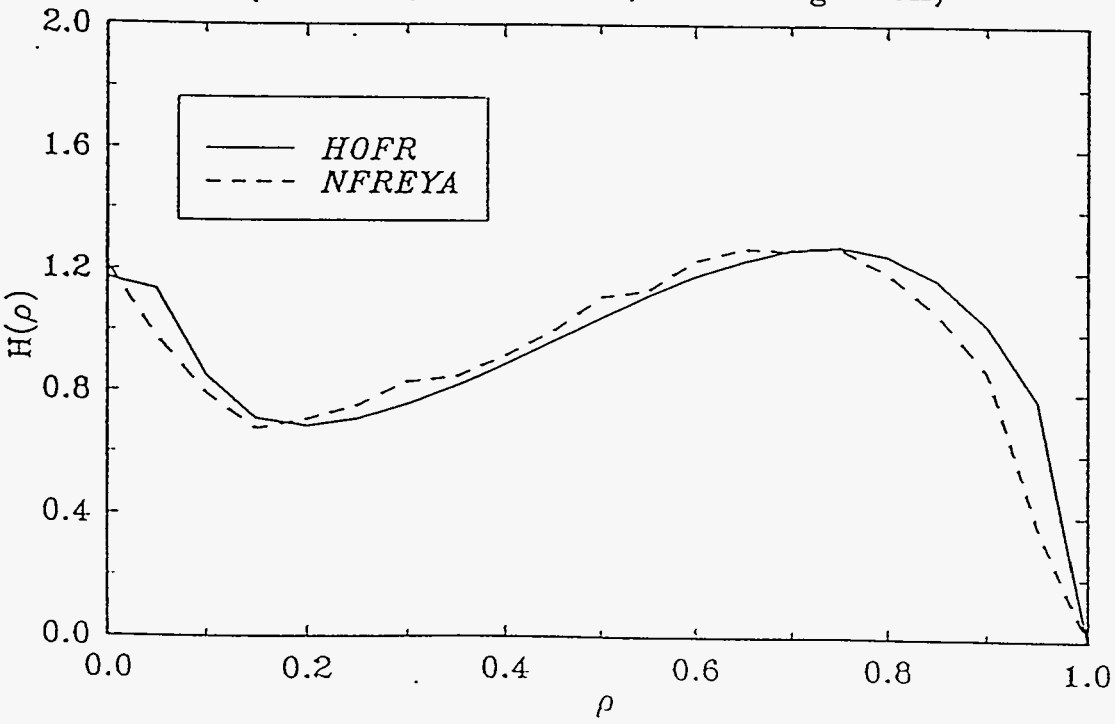
Neutral Beam Benchmark

NB MODULE FOR THE SUPERCODE
(Comparison with the ACCOME code for the B6 SS case)



HOFR and NFREYA comparison

($R_{tan} = R_0$, $E_b = 1.3$ MeV, ITER CDA Ignition)



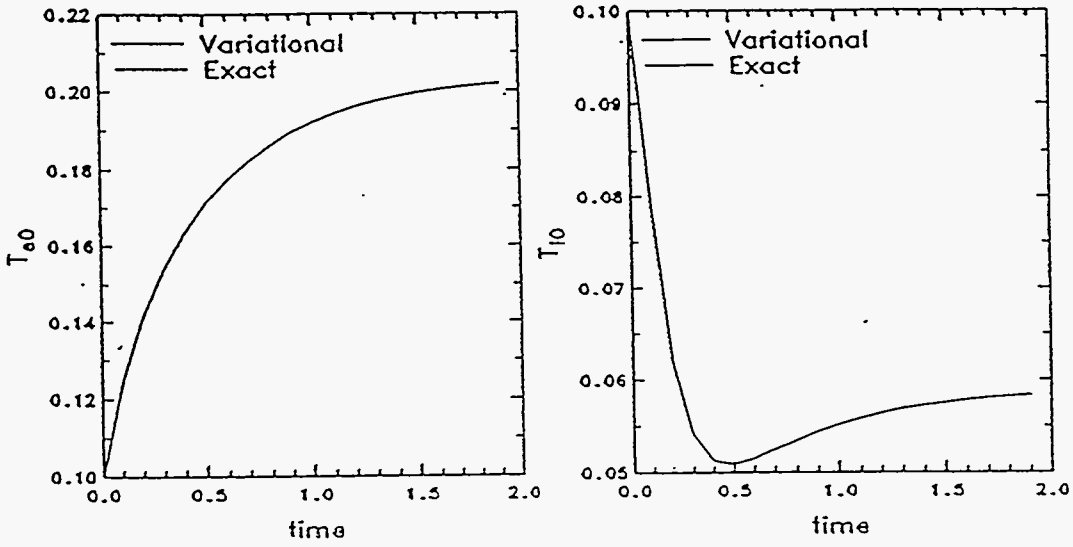
Time-dependent Transport Example

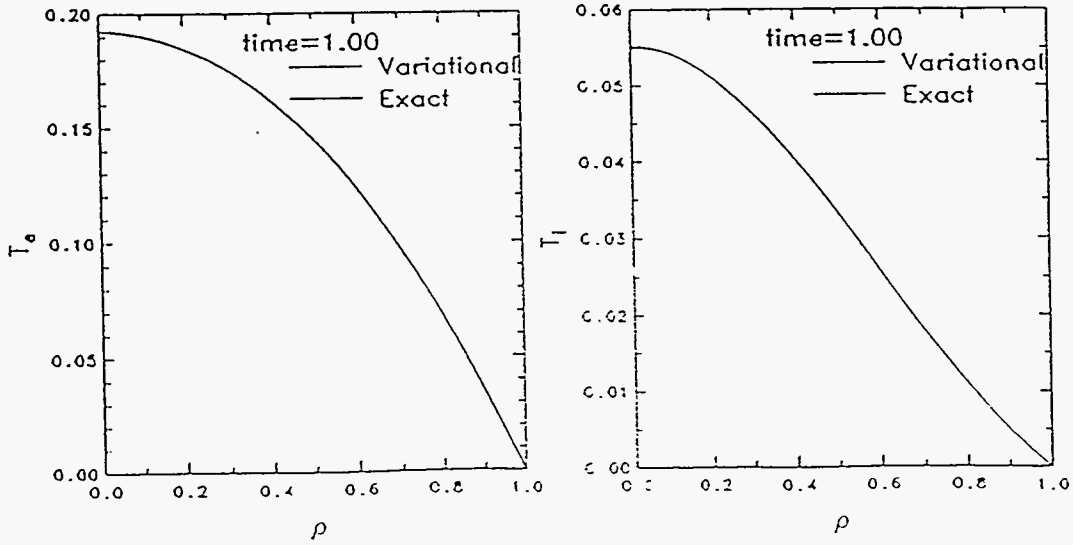
Model Problem:

$$\frac{\partial T_e}{\partial t} = \frac{1}{\rho}\frac{\partial}{\partial \rho}\left(0.1\rho \exp(\rho^2)\frac{\partial T}{\partial \rho}\right) + \frac{1}{(T_e+2)^{1.5}} + 5\frac{T_i - T_e}{(T_e+2)^{1.5}}$$

$$\frac{\partial T_i}{\partial t} = \frac{1}{\rho}\frac{\partial}{\partial \rho}\left(0.5\rho \exp(\rho^2)\frac{\partial T}{\partial \rho}\right) - 5\frac{T_i - T_e}{(T_e+2)^{1.5}}$$

$$T_e(1,t) = T_i(1,t) = 0$$





AN EXAMPLE OF SSAT OPTIMIZATION WITH *SUPERCODE*

OPTIMIZATION

Find the minimum size/cost machine, which obtains:

MISSION (Single-Mode)

high beta first stability, inductive startup and ≥20s burn , steady-state NB current-drive, subject to:-

PHYSICS CONSTRAINTS

Equality constraints: *H=2.0 (ITER-P+DIII), β_N=3.5, $q_\psi(95\%)$=3.0, $q_\psi(0)$=1.0, κ_X=2.0, δ_{95}=0.3, ..., etc*

Inequality constraints: *$\nu^* \leq 0.3$, $\nu_{fast} \cdot \tau_E \geq 8$, $\langle n \rangle_{min} \leq 0.3$Greenwald, $t_{burn} \geq 20s$, $\langle n \rangle_{max} \leq$ min{Greenwald or ITER limit},...., etc*

ENGINEERING CONSTRAINTS

Equality constraints: *radial and vertical builds, ..., etc*

Inequality constraints: *$P_{NB} \leq 30MW$, $P_{NB}+P_{aux} \leq 30MW$, TF and PF magnet allowables (stress, protec., stab., etc) ≤ ITER limits, ..., etc.*

Divertor conditions: *(a) unconstrained, (b) ≤10MW/m^2, (c) ≤5MW/m^2*

SYSTEMS VARIABLES

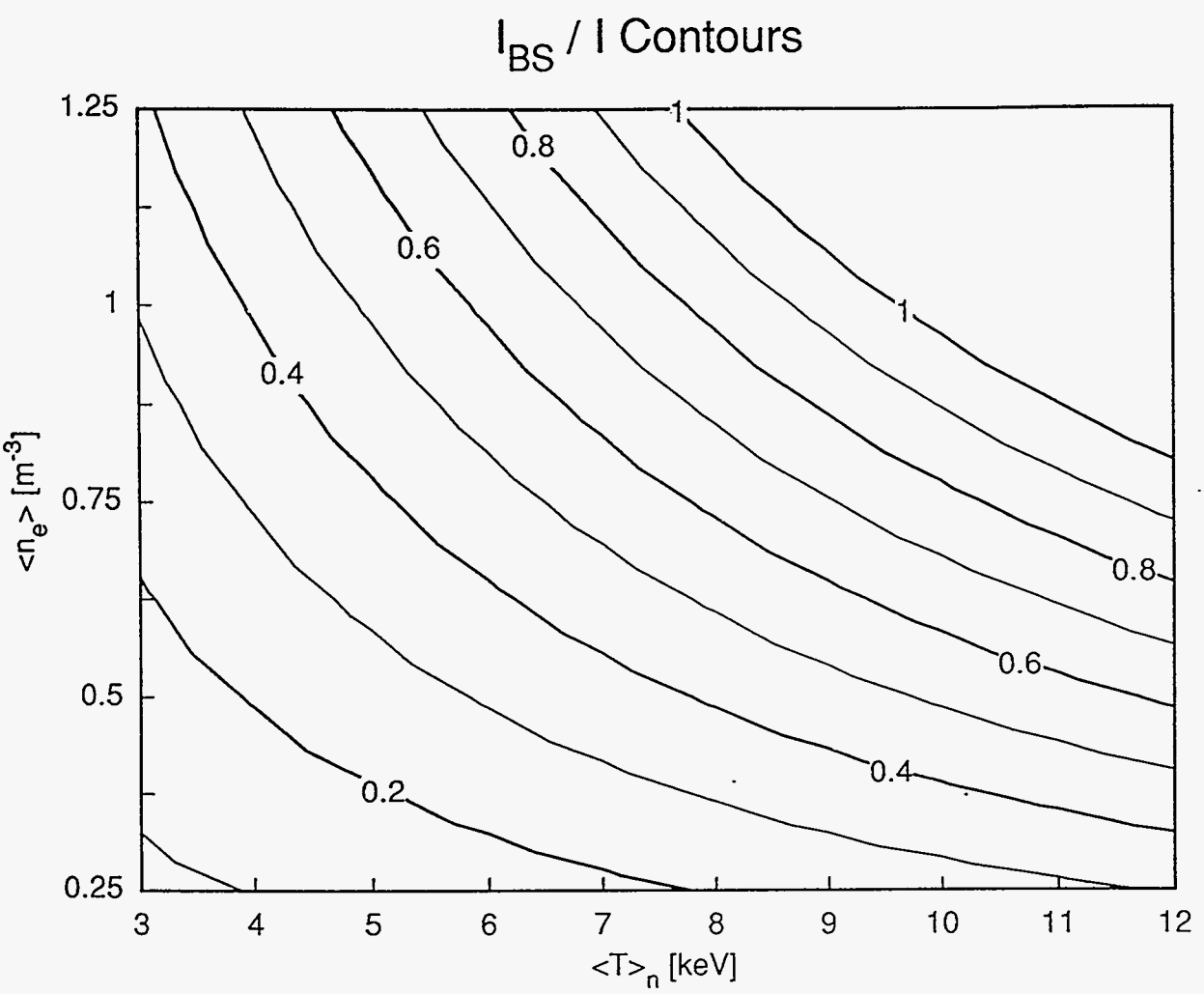
R, a, (A), I, B_o, n, T, f_{BS}, Δ_{TF}, B_{TF}, Δ_{OH}, ..., etc. (Profiles also variable in 1-1/2D)

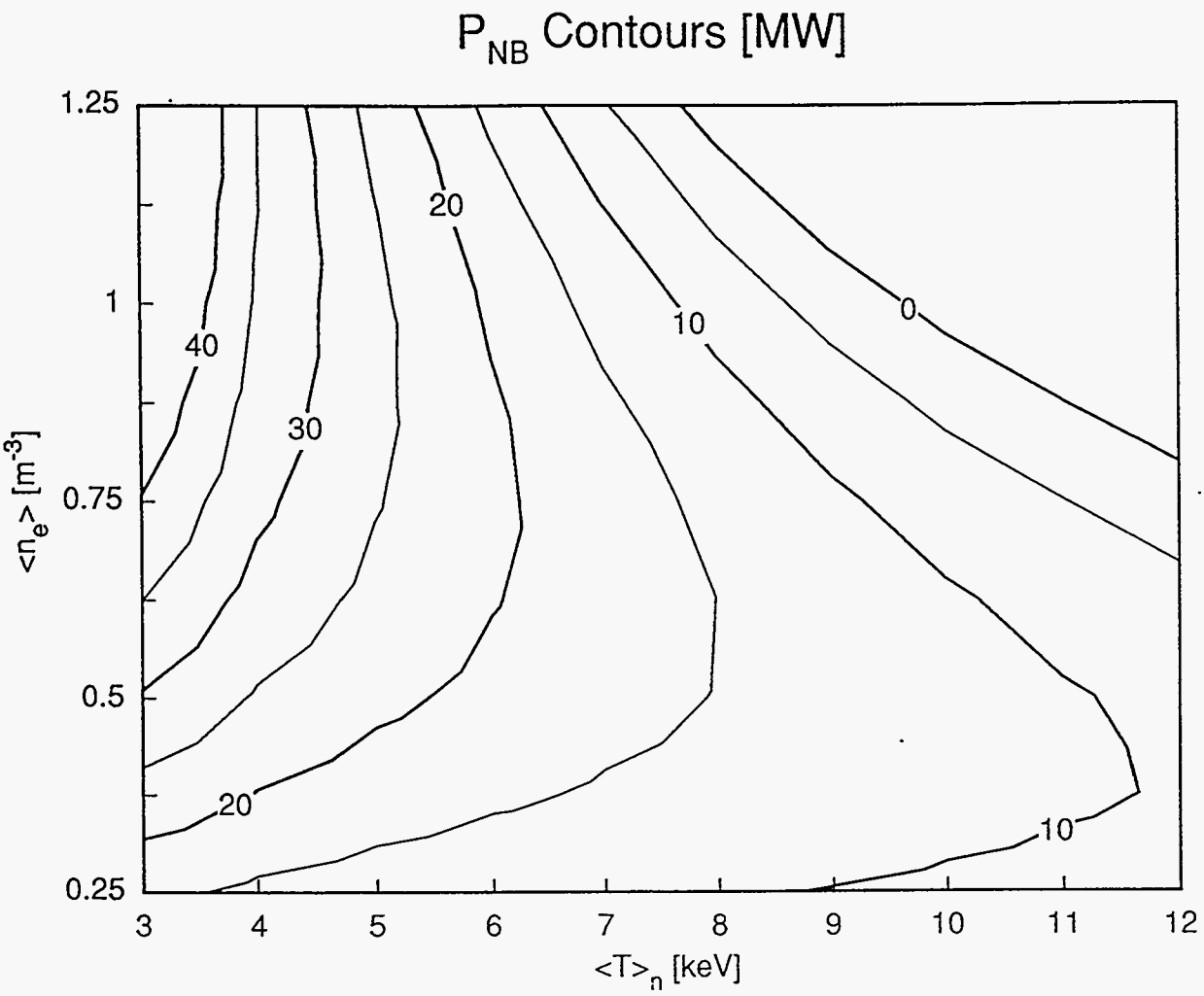
EXAMPLE OF *SUPERCODE* OPTIMIZATION: MINIMUM MAJOR RADIUS MACHINES

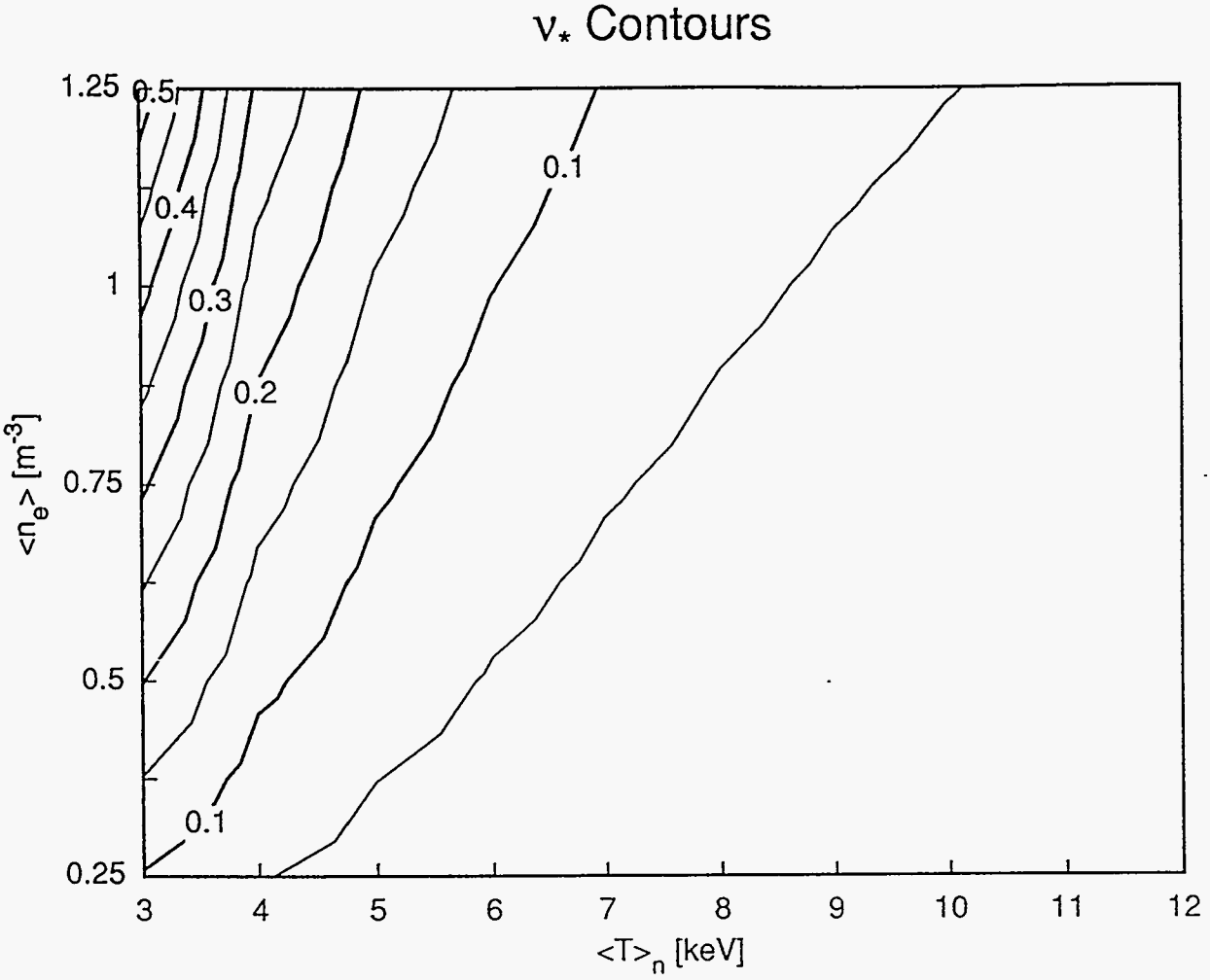
ASPECT RATIO: **DIVERTOR:**	*UNCONSTRAINED* *UNCONSTRAINED*	*A≤5.0* *UNCONSTRAINED*	*A≤4.0* *UNCONSTRAINED*	*UNCONSTRAINED* *≤10MW/m²*	*UNCONSTRAINED* *≤5MW/m²*
R (m) -- minimum!	1.72	1.76	1.85	2.03	2.44
a (m)	0.288	0.352	0.463	0.337	0.416
A	5.96	5.0*	4.0*	6.01	5.86
I (MA)	0.790	1.08	1.65	0.712	0.663
B_o (T)	3.58	3.29	2.95	2.77	2.02
$\langle n_e \rangle$ (e20m^{-3})	0.91*	0.83*	0.73*	0.60*	0.37*
$\langle T_e \rangle_n$ (keV)	3.8	4.3	5.1	3.3	2.8
$\langle n_e \rangle_{max\ Greenwald}$	3.03	2.77	2.44	1.99	1.22
$\langle n_e \rangle_{max\ ITER}$	1.80	1.81	1.80	1.19	0.735
f_{BS}	0.453	0.416	0.370	0.435	0.405
ν^*	0.3*	0.169	0.080	0.3*	0.3*
$\nu_{fast} \cdot \tau_E$	28.9	31.1	34.4	22.1	16.3
B_{TF} (T)	6.76	6.52	6.27	4.84	3.33
Volt-sec:					
total (Vs)	8.84	8.65	11.4	7.54	29.5
startup (Vs)	6.96	6.09	8.22	6.67	27.2
burn (Vs)	1.88	2.56	3.18	0.88	2.4
t_{burn} (s)	30	20*	20*	20*	35*
P_{NB} (MW)	11.7	15.6	22.8	9.16	7.16
P_{aux} (MW)	0	0	0	0	0
Q_{div} (MW/m²)	18.8	20.9	24.0	10.0*	5.0*

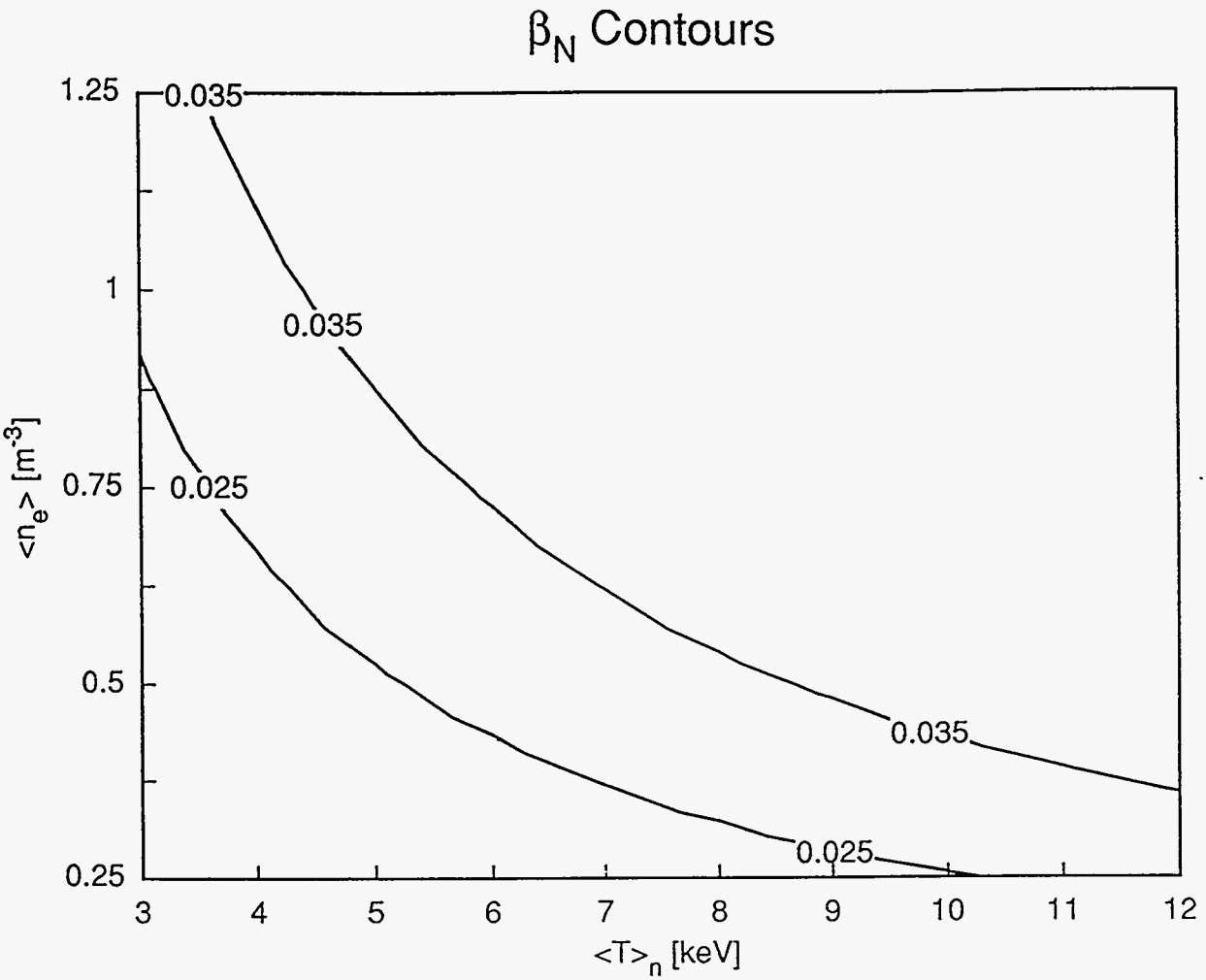
* -- Variable at a constraint bound

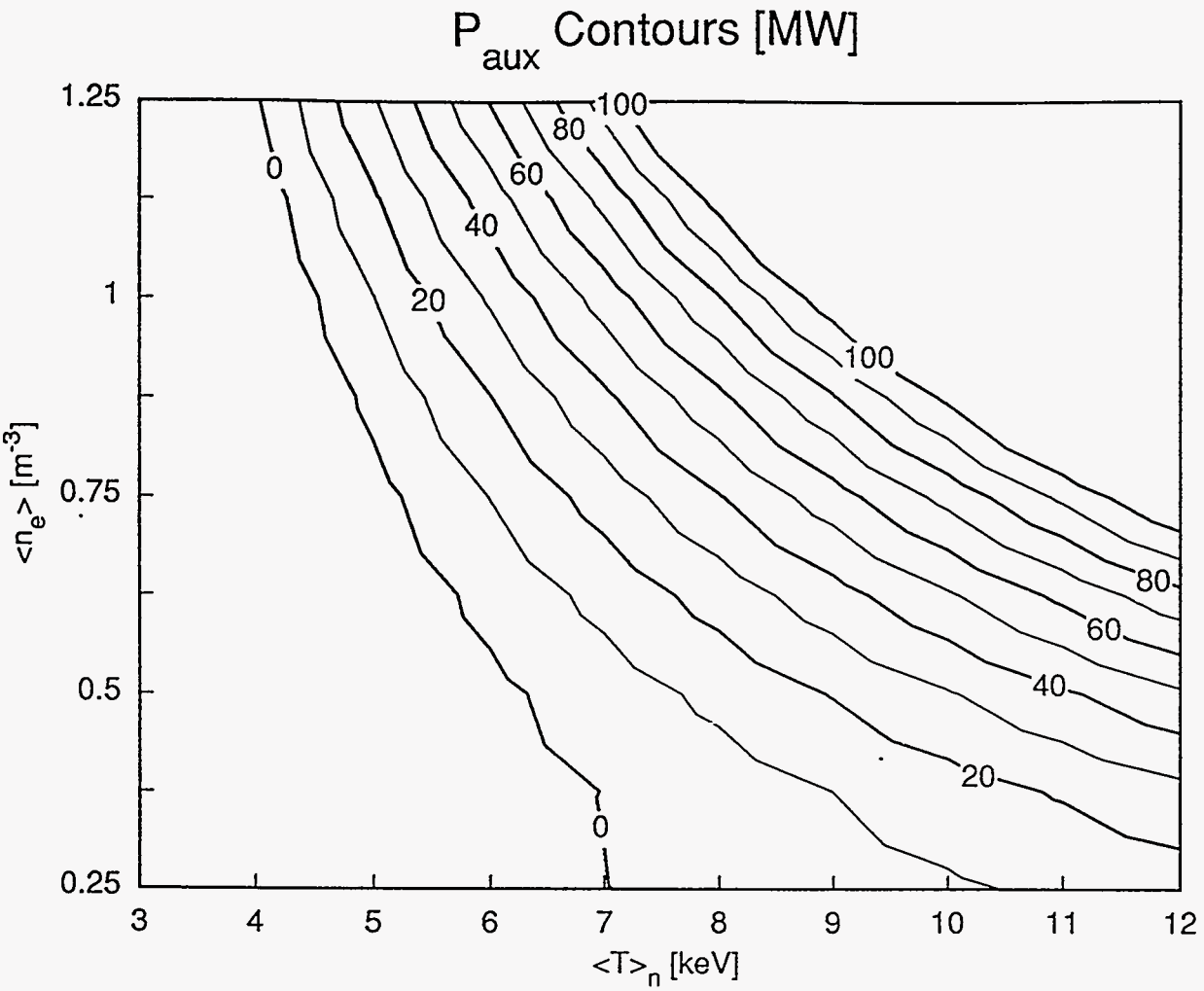
EXAMPLE OF POPCON OPERATING SPACE CONTOURS FROM *SUPERCODE*: SSAT @ 3.35T, 1.66MA

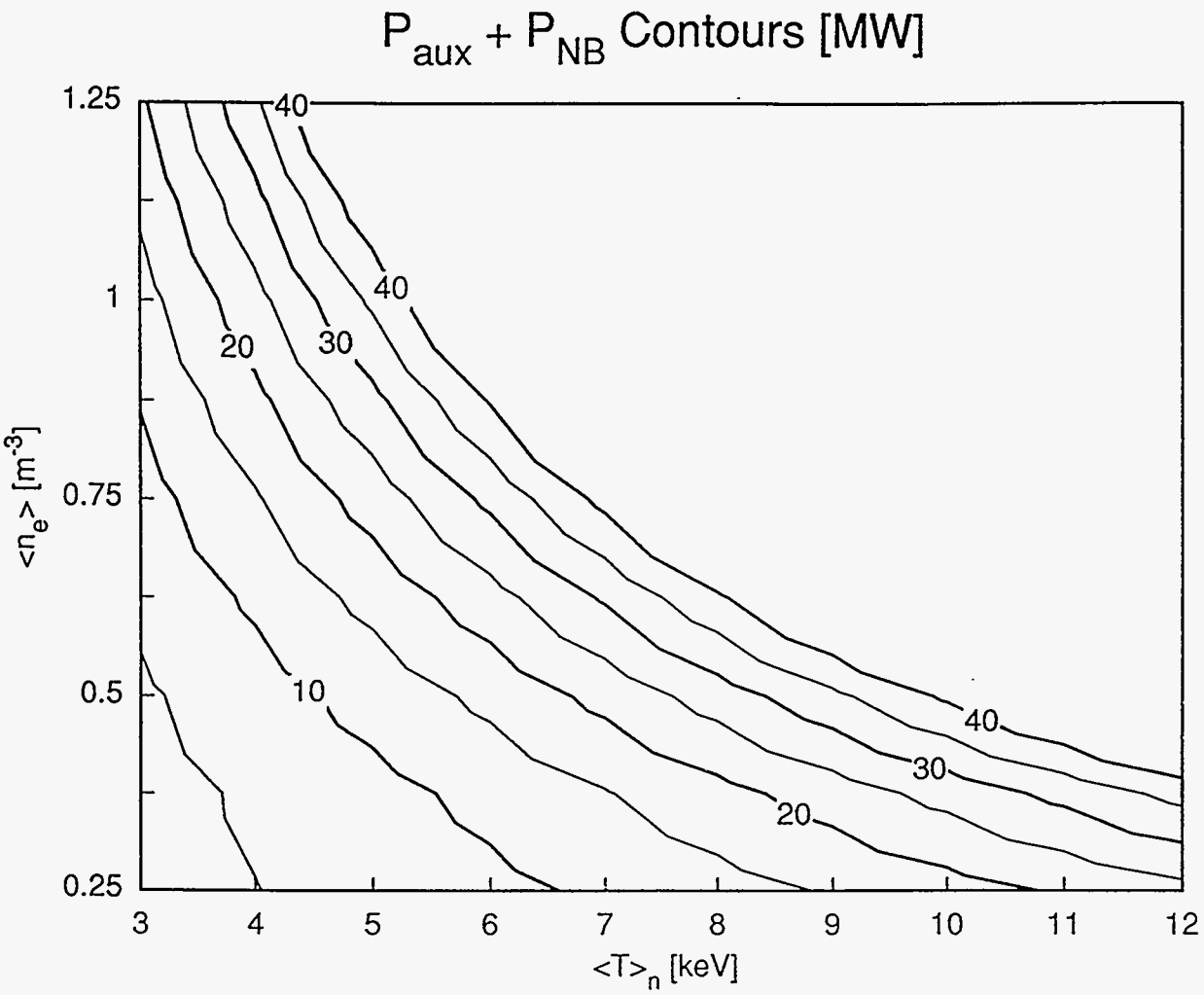

I_{BS} / I Contours
1.25
1
0.75
0.5
0.25
$<n_e>$ [m^{-3}]
3
4
5
6
7
8
9
10
11
12
$<T>_n$ [keV]
1
0.8
0.6
0.4
0.2

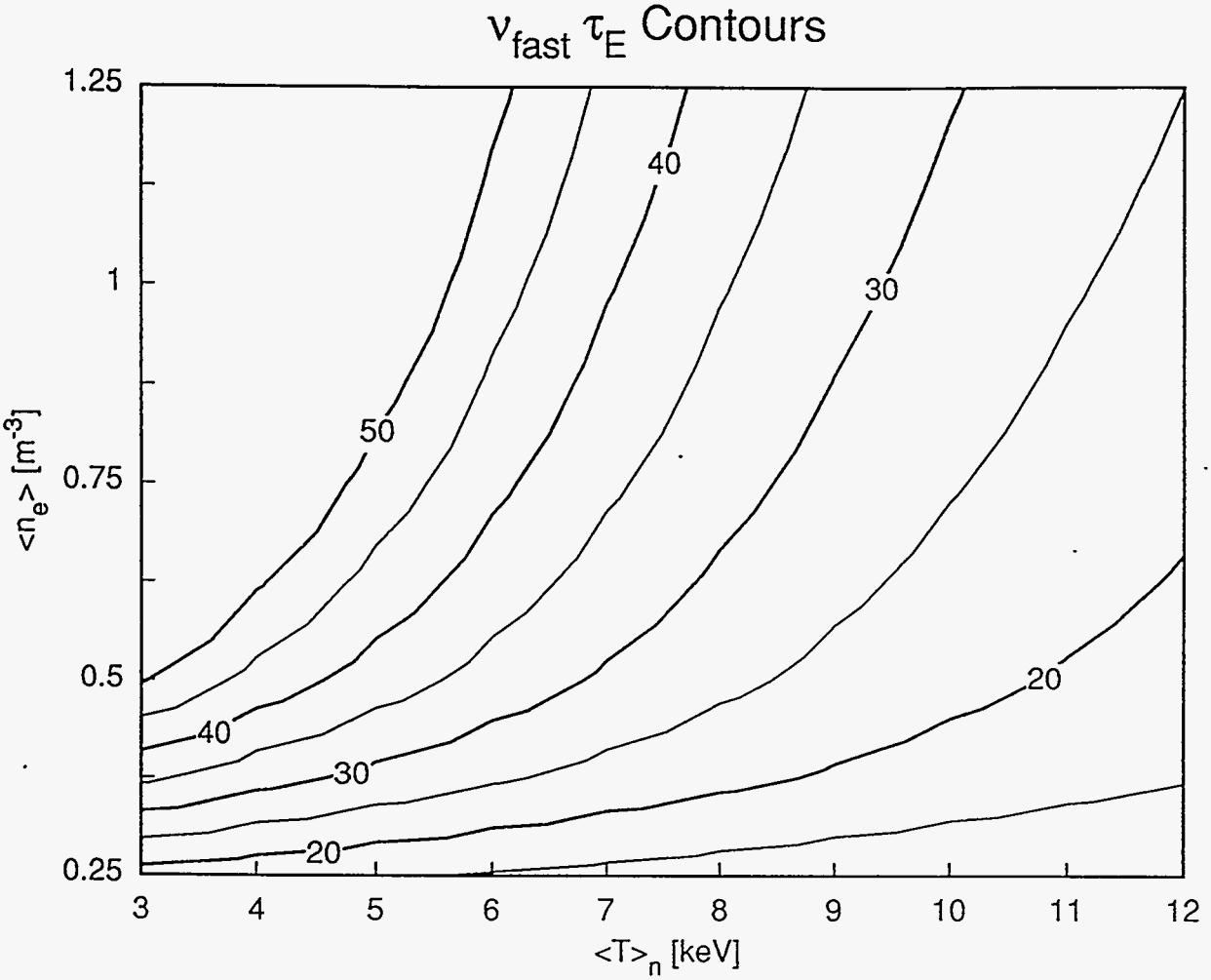

P_NB Contours [MW]
1.25
1
0.75
0.5
0.25
<n_e> [m^-3]
3
4
5
6
7
8
9
10
11
12
<T>_n [keV]
40
30
20
20
10
10
0

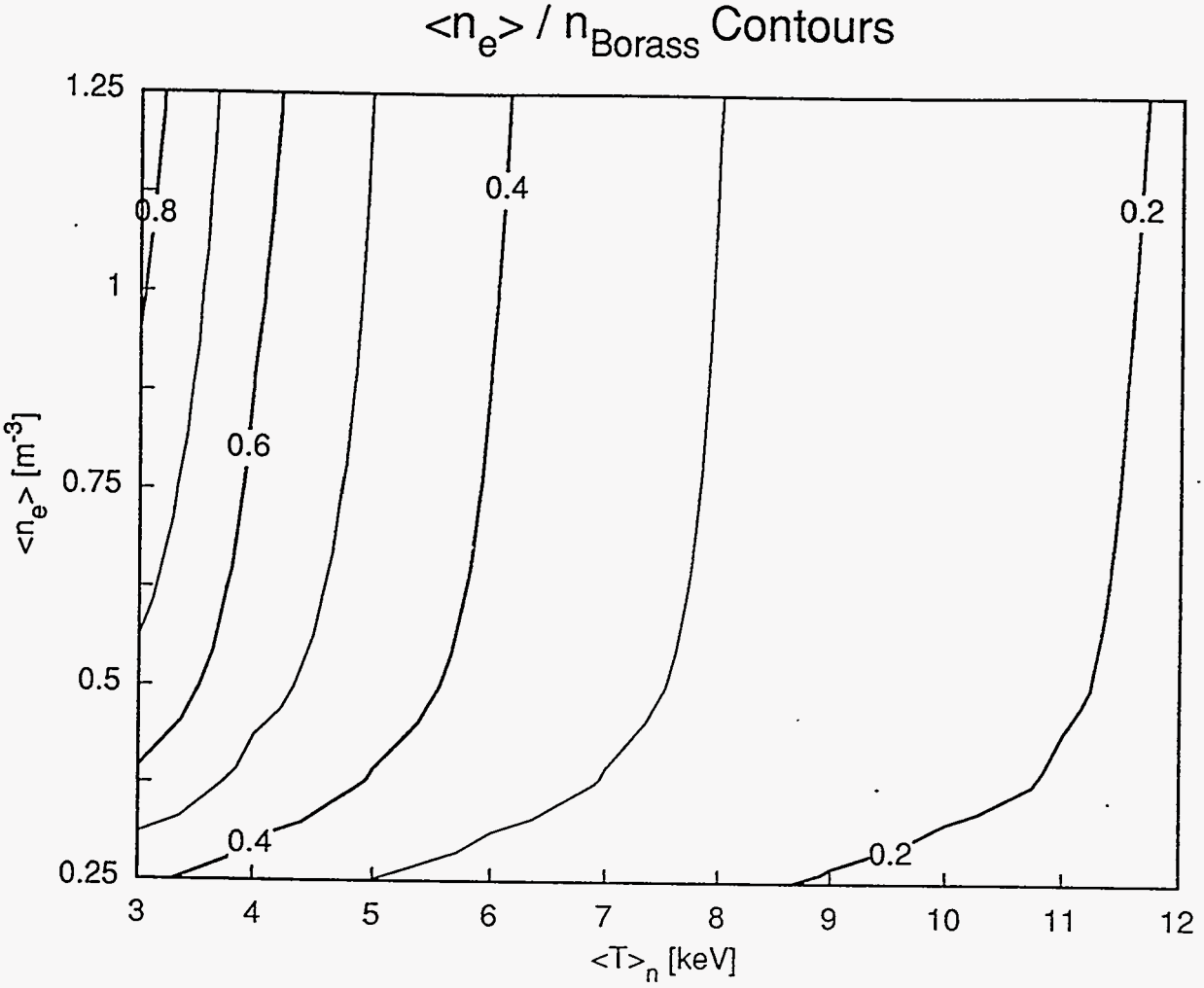

ν* Contours
1.25
1
0.75
0.5
0.25
<ne> [m-3]
0.5
0.4
0.3
0.2
0.1
0.1
3
4
5
6
7
8
9
10
11
12
<T>n [keV]

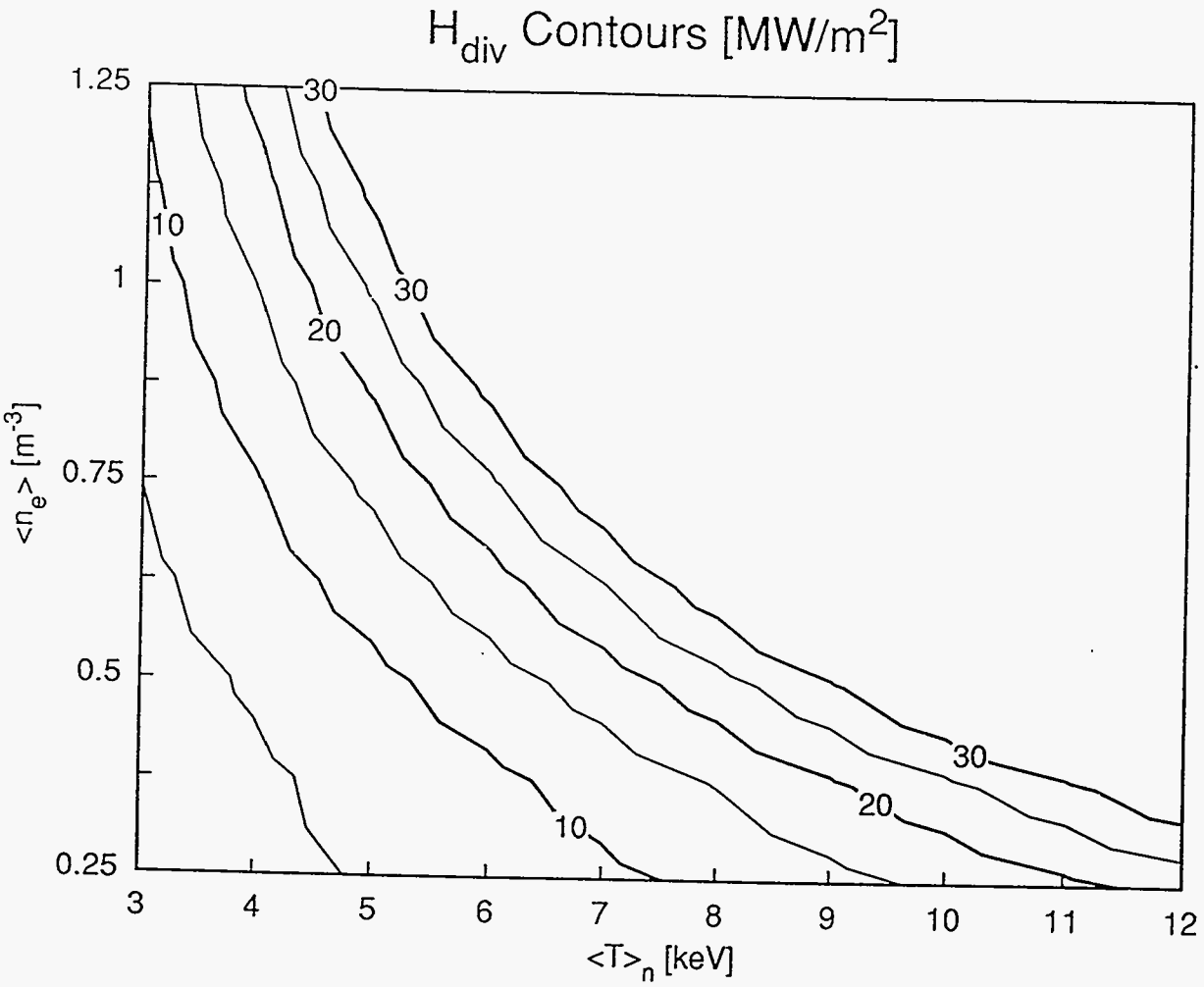

β_N Contours
1.25
1
0.75
0.5
0.25
$<n_e>$ [m^{-3}]
0.035
0.035
0.025
0.035
0.025
3
4
5
6
7
8
9
10
11
12
$<T>_n$ [keV]


P aux Contours [MW]
1.25
1
0.75
0.5
0.25
<n_e> [m^-3]
3
4
5
6
7
8
9
10
11
12
<T>_n [keV]
0
20
40
60
80
100
100
80
60
40
20
0


P aux + P NB Contours [MW]
1.25
1
0.75
0.5
0.25
<n e> [m-3]
40
40
30
20
10
40
30
20
3
4
5
6
7
8
9
10
11
12
<T> n [keV]


$\nu_{fast}\ \tau_E$ Contours
1.25
1
0.75
0.5
0.25
$<n_e>$ [m^{-3}]
3
4
5
6
7
8
9
10
11
12
$<T>_n$ [keV]
50
40
40
30
30
20
20


<n_e> / n_Borass Contours
1.25
1
0.75
0.5
0.25
<n_e> [m^-3]
0.8
0.6
0.4
0.2
0.4
0.2
3
4
5
6
7
8
9
10
11
12
<T>_n [keV]


H_{div} Contours [MW/m^2]
1.25
1
0.75
0.5
0.25
<n_e> [m^{-3}]
3
4
5
6
7
8
9
10
11
12
<T>$_n$ [keV]
30
10
30
20
30
20
10

Exploratory *SUPERCODE* Runs Completed for TPX

- Maximum f_{BS}, β_N and n.τ.T runs for 3.35T baseline under:
  (a) Day 1 conditions (Pnb=16MW, Pic=12MW)
  (b) Upgrade conditions (Pnb=32MW, Pic=18MW)
  (all plasma parameters varying subject to physics "rules")

- SSAT 3.35 baseline: Minimum cost TF coil optimization (in association with J.Schultz)

- Minimum cost machines for:
  (a) SSAT ST scenario (β_N=3.5% @ q_{95}=3.0)
  (b) SSAT AT scenario (f_{BS}=0.667 @ β_N=3.0%)
  (all plasma, geometry, and build parameters varying subject to physics "rules" and engineering constraints.)

- Survey of cost -v- performance of SSAT-like machines in A-$ space ($350 \leq \$ \leq 500M, $3.0 \leq A \leq 7.0$)

- Possible lower cost, higher performance design points selected from this A-$ space

- SSAT baseline: reduction of f_{BS} and β_N subject to:
  (a) Divertor peak heat load constraint
  (b) Divertor temperature constraint

- Preliminary benchmark of JT-60SU

- SSAT baseline: 1-1/2-D study of beam tangency relative to port access, coil geometry, current-profiles and plasma performance.

- Cost impact of SN $\rightarrow$ DN change for baseline (~$8M, i.e ~2%)

- SSAT baseline: Cost / performance impact of $B_0 = 3 \rightarrow 6$ Tesla (in progress)

Typical Applied Constraints

PHYSICS AND OPERATIONAL CONSTRAINTS

- β_N =3.5 (ST)/3.0(AT), • H = 2.0 (quadrature DIII and $H\tau_{ITER-P}$), • κ_X = 2.0,
- $V^* \leq 0.3$ • Coll. correction on 1-D bootstrap model.,
- $V_{fast}\,\tau_E \geq 8$, • $\langle n \rangle \leq I/\pi a^2$ (Greenwald),
- $\langle n \rangle \geq n_{NB} = (dN_{NB}/dt).2\tau_E/(1 - 0.5)V$, • $\langle n \rangle \geq 0.3$ e20m^{-3}
- V•sec = full inductive rampup (Cejima = 0.6) + 2V-sec for flattop,
- For AT: $1.05 \leq q_0 \leq 1.30$ (first stability regime) and $q_{95} \geq 3.0$,
- For ST: q_0 =1.05 and q_{95} = 3.0, • Separate electron and ion power bal.
- $\beta_{fast}/\beta \leq 0.25$, • $\varepsilon^*\beta_p \leq 1.0$ (first stability regime).
- New, density-dependent impurity specs
- $R_{tan,NB}$ = R - a/2, except where noted,
- $I = I_{NB} + I_{IC} + I_{BS}$, where I_{IC} can be <0 if req'd.
- $P_{NB} \leq$ 16 MW and $P_{IC} \leq$ 12 MW (for baseline, day 1 runs),
- $-1 \leq f_{IC} \leq 1$, where $I_{IC} = f_{IC}\,P_{IC}\,\gamma_{IC}/nR$.
- Divertor conditions: unconstrained, except where noted.
- α_n= 0.5, α_T= 1.0 (variable in 1-1/2D if χ's supplied).

GENERAL ENGINEERING CONSTRAINTS

Optimization runs have engineering design constraints for:
PF, TF, builds, ripple, port access,.... ,etc. (examples shown later.)
Eng. variables for baseline runs --stresses, builds, etc. -- are fixed.

SYSTEMS VARIABLES FOR OPTIMIZATION RUNS

R, a, (A), I, q0, q95, B_o, n_e, n_i, T_e, T_I, f_{BS}, P_{NB}, P_{IC}, f_{IC}, Δ_{TF}, B_{TF}, Δ_{OH}, builds, etc. . . .(Profiles also variable in 1-1/2 D.)

SSAT 3.35T BASELINE: OPTIMIZED PLASMA PERFORMANCE -- DAY 1 SPECIFICATIONS

	Maximum bootstrap fraction @β_N=3.0%	Maximum β_N @q95=3.0	Maximum $n_i(0).\tau E.T_i(0)$ @$\beta_N \leq 3.5\%$
Bootstrap fraction	0.639 (max)	0.349	0.361
β_N (%)	3.0*	3.14 (max)	2.99
$n_i(0).\tau E.T_i(0)$ (10^{20} keV.s.m^{-3})	0.202	1.30	1.35(max)
I (MA)	0.435	1.75*	1.77*
q_0	1.30*	1.05*	1.3*
q_{95}	7.90	3.0*	3.0*
T_e (keV)	5.10	7.17	6.39
T_i (keV)	4.13	9.21	8.07
n_e (10^{20} m^{-3})	0.3*	0.481*	0.537
τ_E (s)	0.075	0.132	0.139
P_{NB} (MW)	5.19	16.0*	16.0*
P_{IC}(MW)	12.0*	12.0*	10.4
f_{IC}	-0.227	+0.132	+0.382
$\varepsilon^*\beta_p$	1.0*	0.406	0.384
Peak div. heat load (MW/m^2)	7.23	17.8	16.1
Div. temp. (eV)	551	354	291

* – Variable at a constraint bound

SSAT 3.35T BASELINE: OPTIMIZED PLASMA PERFORMANCE -- UPGRADED SPECIFICATIONS

	Maximum bootstrap fraction @β_N=3.0%	Maximum β_N @q95=3.0	Maximum $n_i(0).\tau_E.T_i(0)$ @$\beta_N \leq 3.5\%$
Bootstrap fraction	0.779 (max)	0.476	0.666
β_N (%)	3.0*	4.37(max)	3.5*
$n_i(0).\tau_E.T_i(0)$ (10^{20} keV.s.m^{-3})	0.164	1.62	1.53(max)
I (MA)	0.681	1.79*	1.11*
q_0	1.30*	1.05*	1.3*
q_{95}	7.86	3.0*	3.0*
T_e (keV)	6.46	6.41	4.78
T_i (keV)	3.69	8.01	6.11
n_e (10^{20} m^{-3})	0.3*	0.805*	0.849
τ_E (s)	0.068	0.110	0.130
P_{NB} (MW)	3.01	32.0*	28.7
P_{IC}(MW)	18.0*	18.0*	5.41
f_{IC}	-0.074	-0.352	+1.0*
$\varepsilon^*\beta_p$	1.0*	0.552	0.449
Peak div. heat load (MW/m^2)	9.25	32.1	19.0
Div. temp. (eV)	659	352	204

* – Variable at a constraint bound

MINIMUM COST MACHINES (Light Bulbs?)

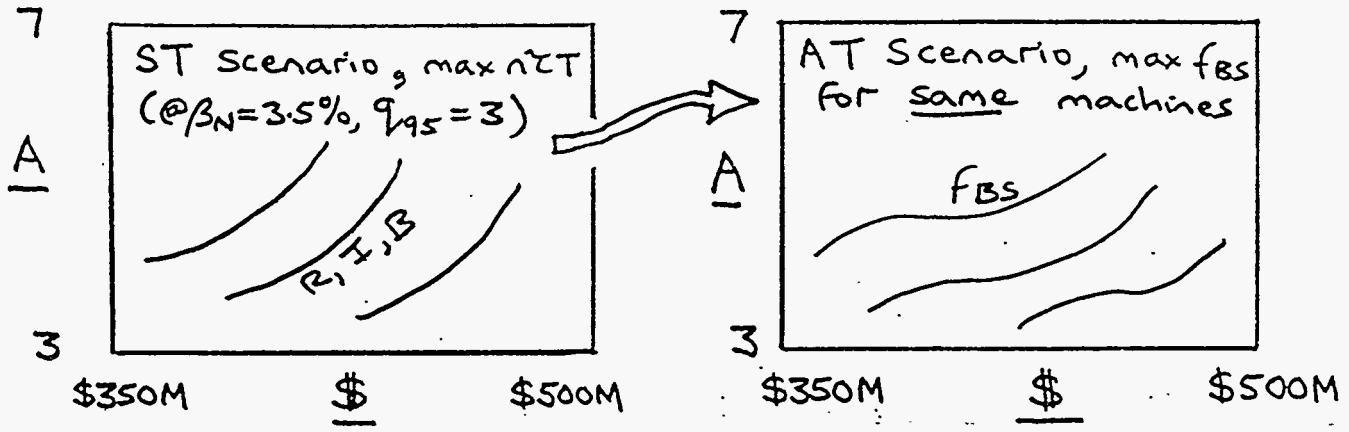
	SSAT Baseline[a]	Min. cost machine for ST scenario: β_N=3.5%, q_{95}=3.0	Min. cost machine for AT scenario: f_{BS}=0.667, β_N=3.0%
Cost (M$)	455	286	296
R (m)	2.25	1.62	1.66
A	4.5	7.32	7.24
I (MA)	0.435 - 1.75	0.319	0.320
q_{95}	3.0 - 7.90	3.0*	3.90
B (T)	3.35	2.34	2.90
T_e (keV)	5.1 - 7.2	3.18	3.72
T_i (keV)	4.1 - 92	1.61	1.76
n_e (10^{20} m^{-3})	0.3 - 0.48	0.629	0.565
β_N(%)	3.0 - 3.14	3.5*	3.0*
$n_i(0).\tau_E.T_i(0)$ (10^{20} keV.s.m^{-3})	0.20 - 1.35	0.074	0.079
Bootstrap fraction	0.35 - 0.64	0.599	0.667*
P_{NB} (MW)	5 - 16.0	0* (lower)	0* (lower)
P_{IC}(MW)	12.0	6.38	6.55
Peak div. heat load (MW/m^2)	7.2 - 18	10.9	9.6
Div temp (eV)	354 - 551	98	131
$P_{div, Neilson}$ (MW/m)	1.2 - 2	0.626	0.628
$\nu_{fast}.\tau_E$ constr. hit?	no	yes*	yes*
ν^* const. hit?	no	yes*	yes*

(a) - Range of parameters for both AT and ST scenarios * – Variable at a constraint bound

A - $ SPACE FOR PERFORMING SSAT DESIGN, COST AND PERFORMANCE TRADEOFFS

- Pick 2 dimensions to display parametric results

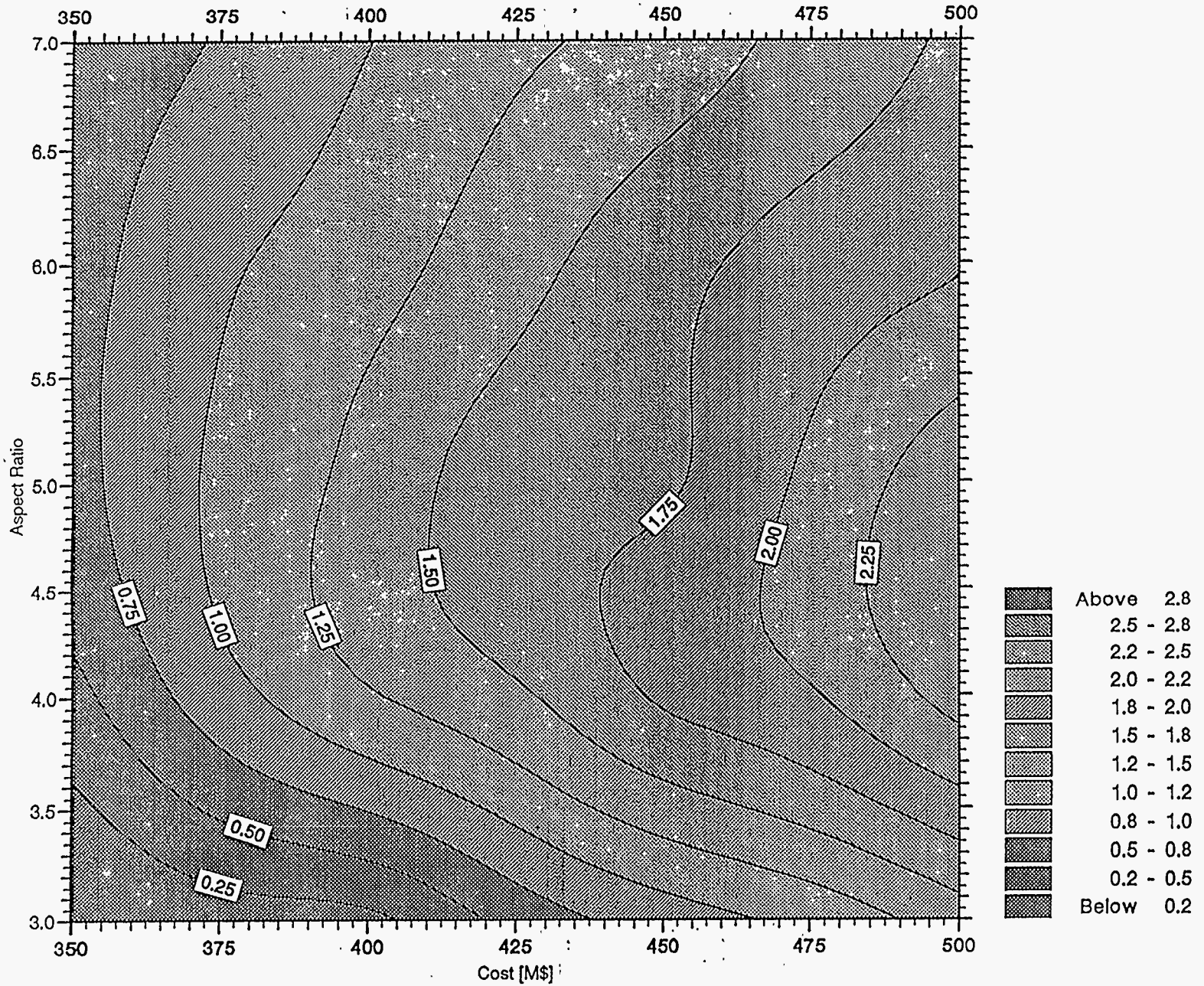
Aspect Ratio vs. Cost



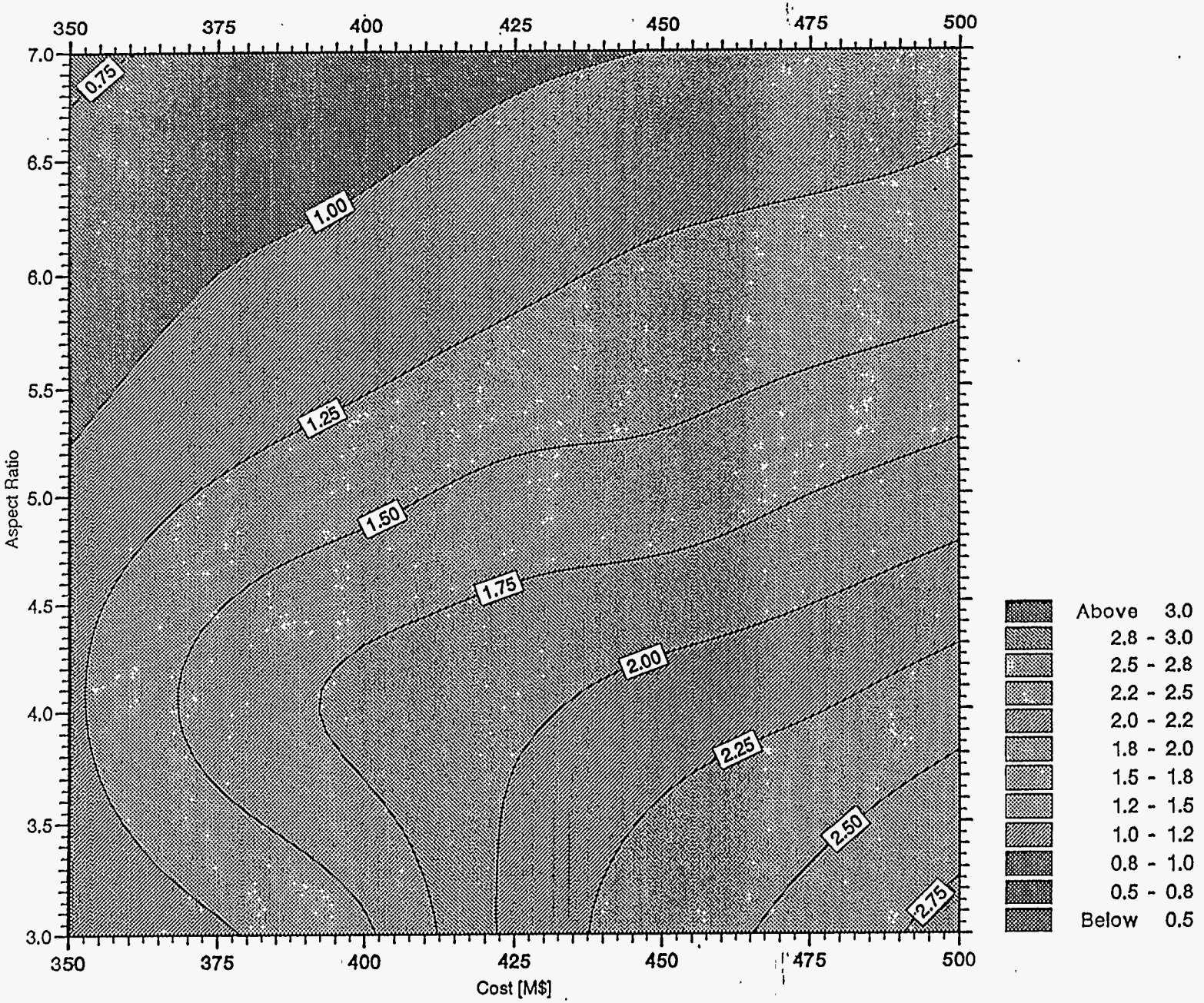
- There are still many other major device parameters to be determined: B_t, I (or q_{95}), n/T , Pinject ...etc

- Use the following method to determine these.
  - Require ST scenario attainment(β_N=3.5% @ q_{95}=3.0) at every [A,$] point
  - Maximize n.τ.T under ST scenario subject to cost constraint and all other physics rules. All design parameters (R, a, I, B, n, P_{aux},...etc) can vary to maximize n.τ.T at [A,$] point
  - This determines a unique set of machines

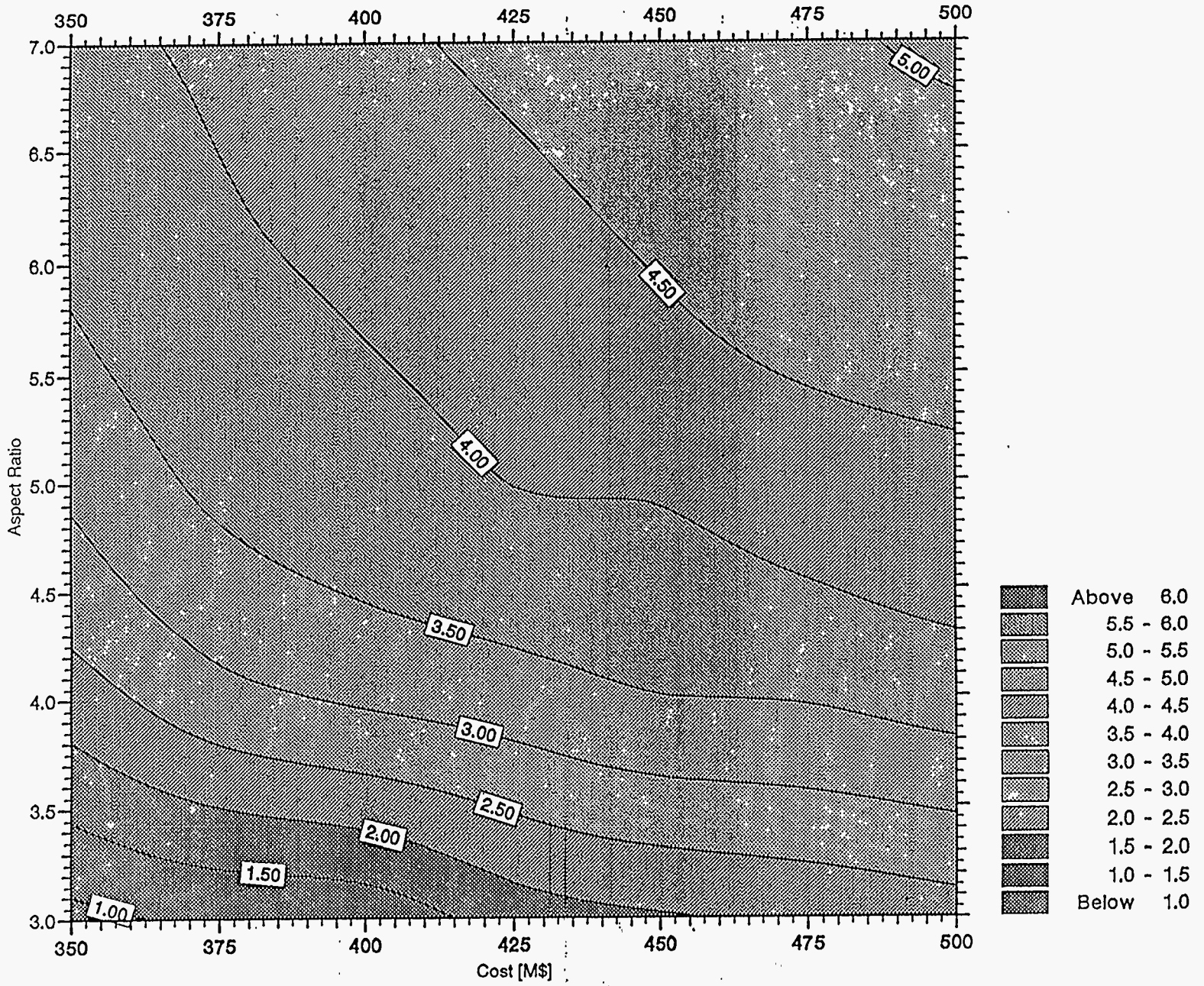
- Then for the same set of machines (same R, a, B,...):
  - Determine the maximum bootstrap fraction possible for these machines under AT scenario (max f_{BS}@β_N=3.0%)
  - Plasma performance variables (n, T, q, P_{aux}...) can vary subject to $I \leq I_{ST}$ (i.e, $q_{95} \geq 3.0$), P_{NB},$P_{IC} \leq$ max installed powers from ST scenario above



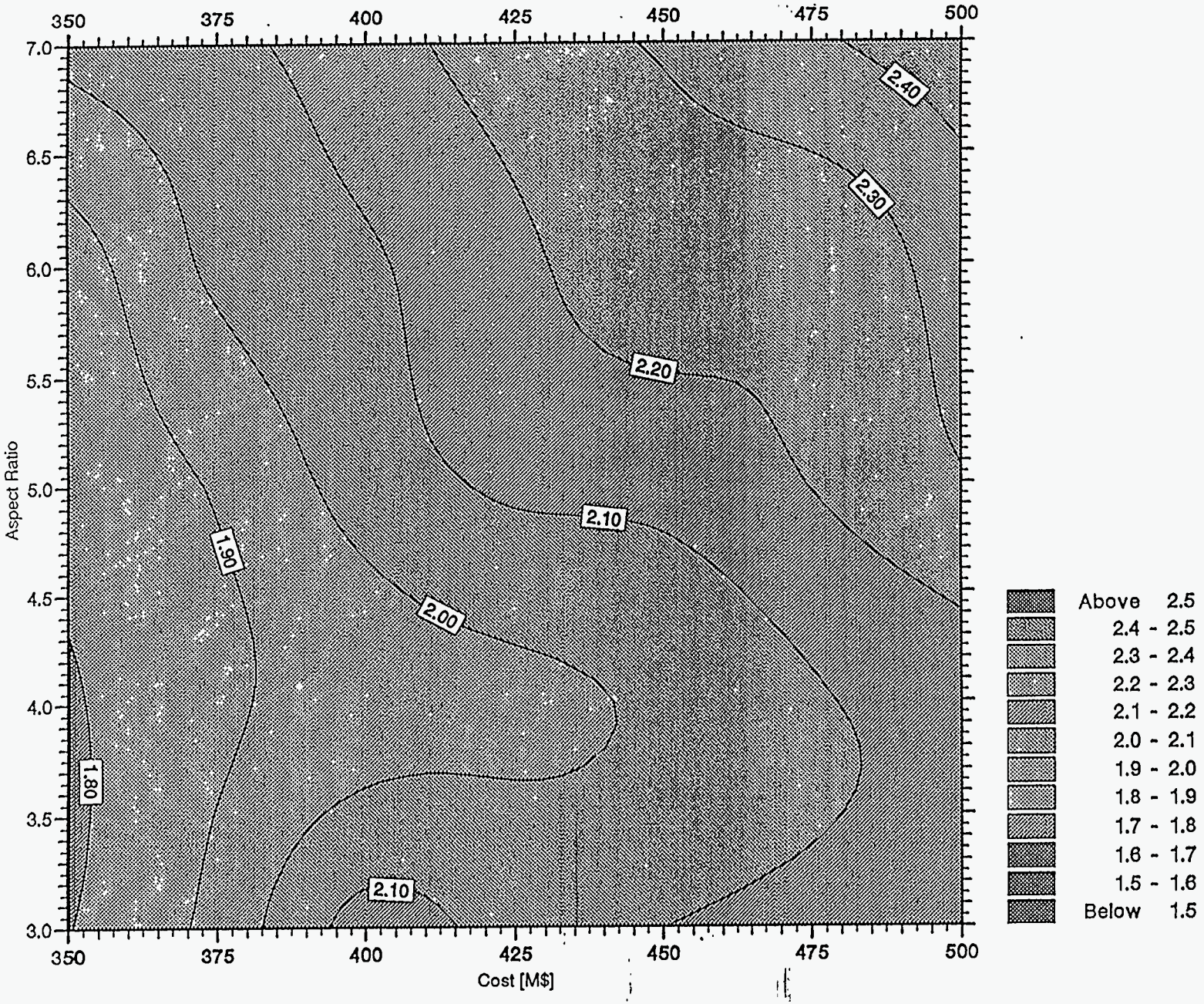
T-n-tau vs Cost and Aspect Ratio



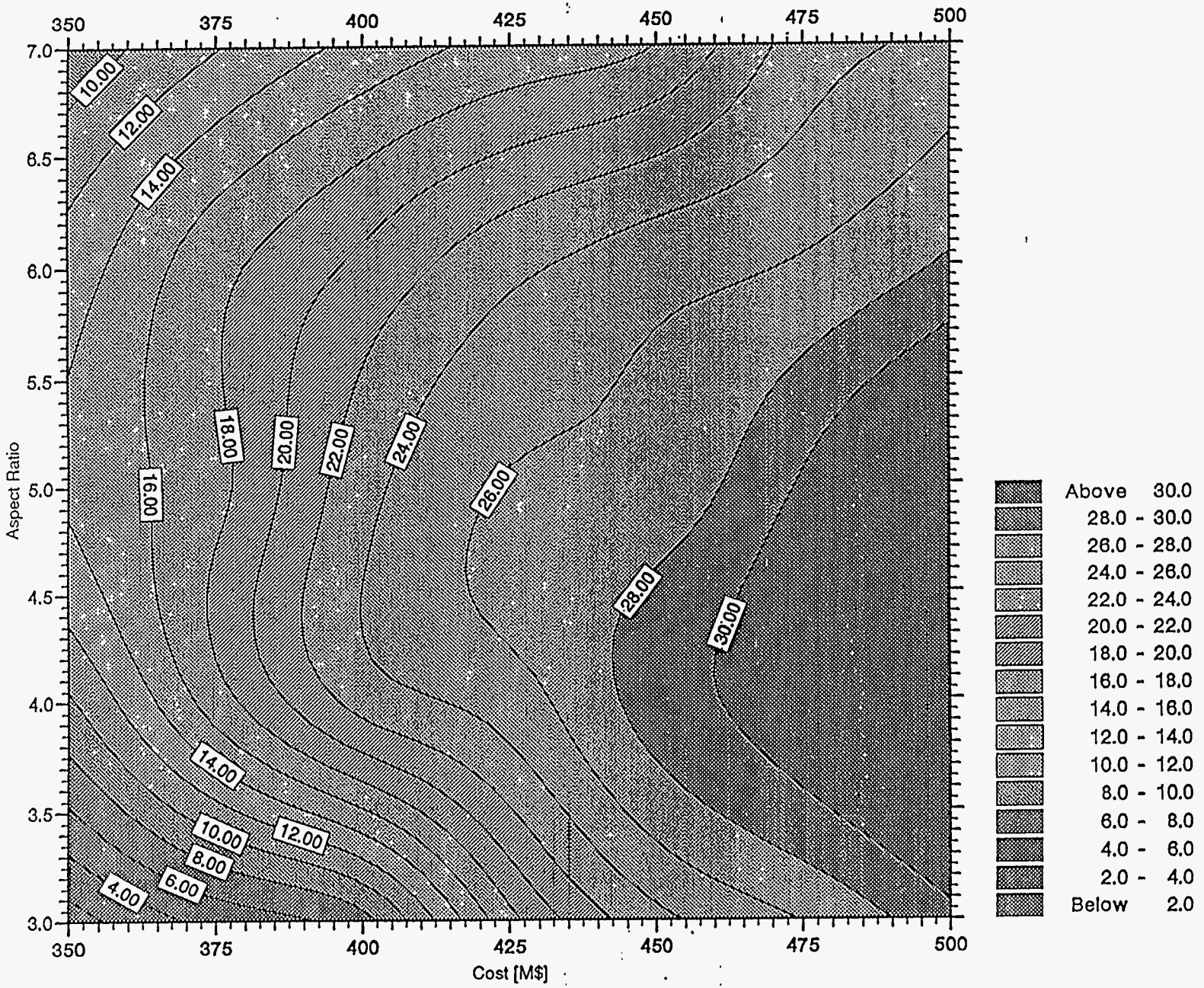
Plasma Current [MA] vs Cost and Aspect Ratio



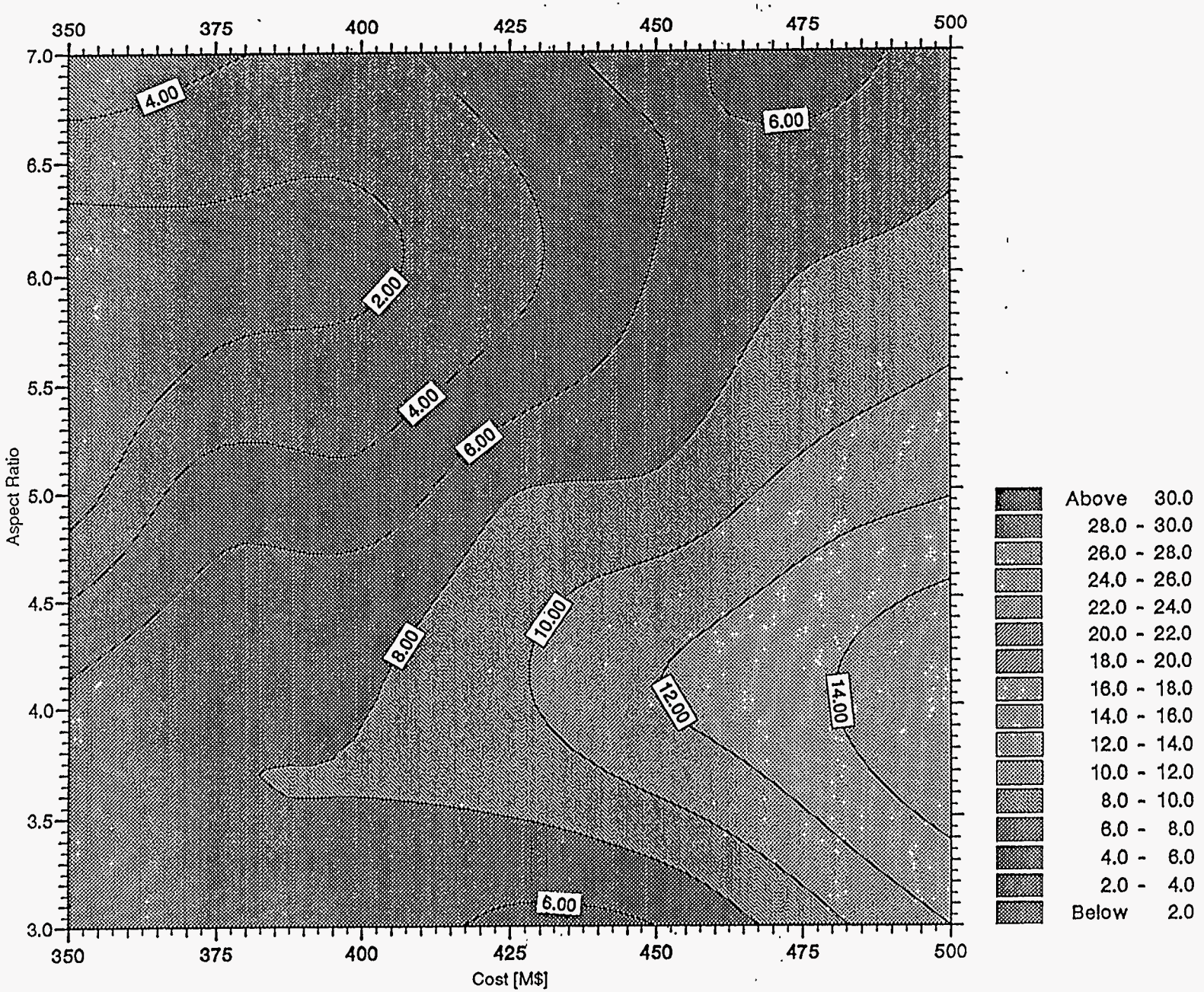
B [T] vs Cost and Aspect Ratio



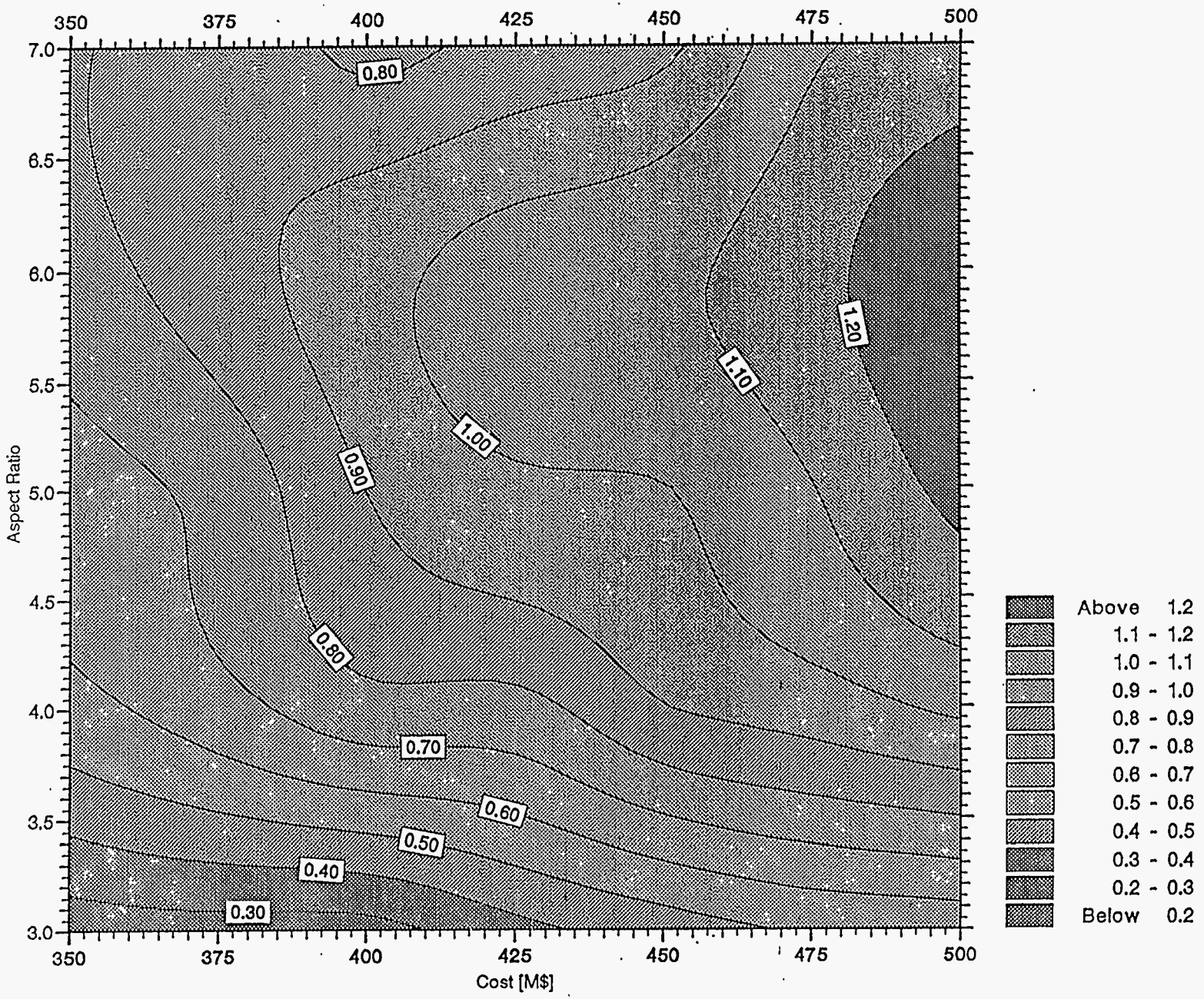
Major Radius [m] vs Cost and Aspect Ratio



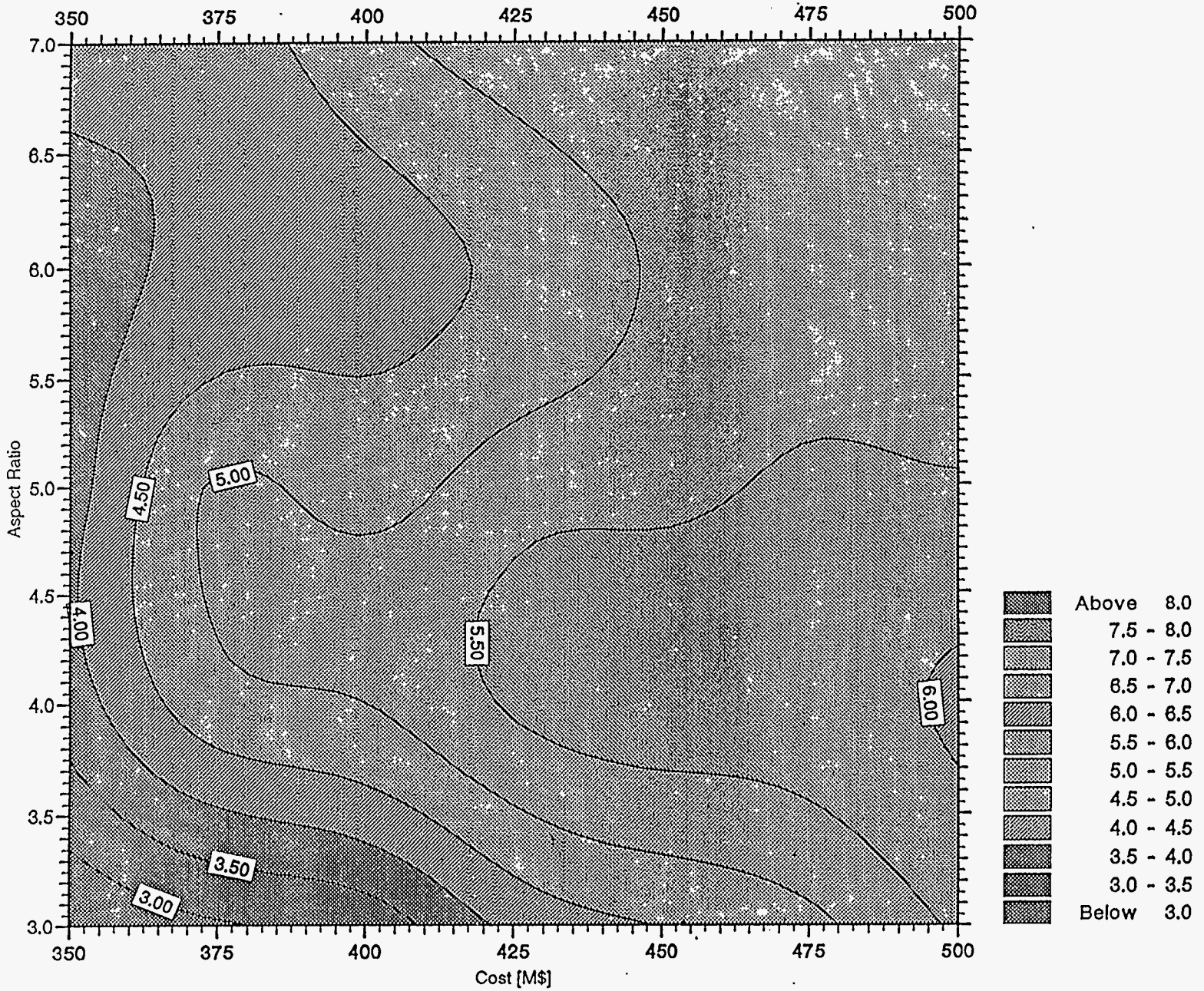
P_NB [MW] vs Cost and Aspect Ratio



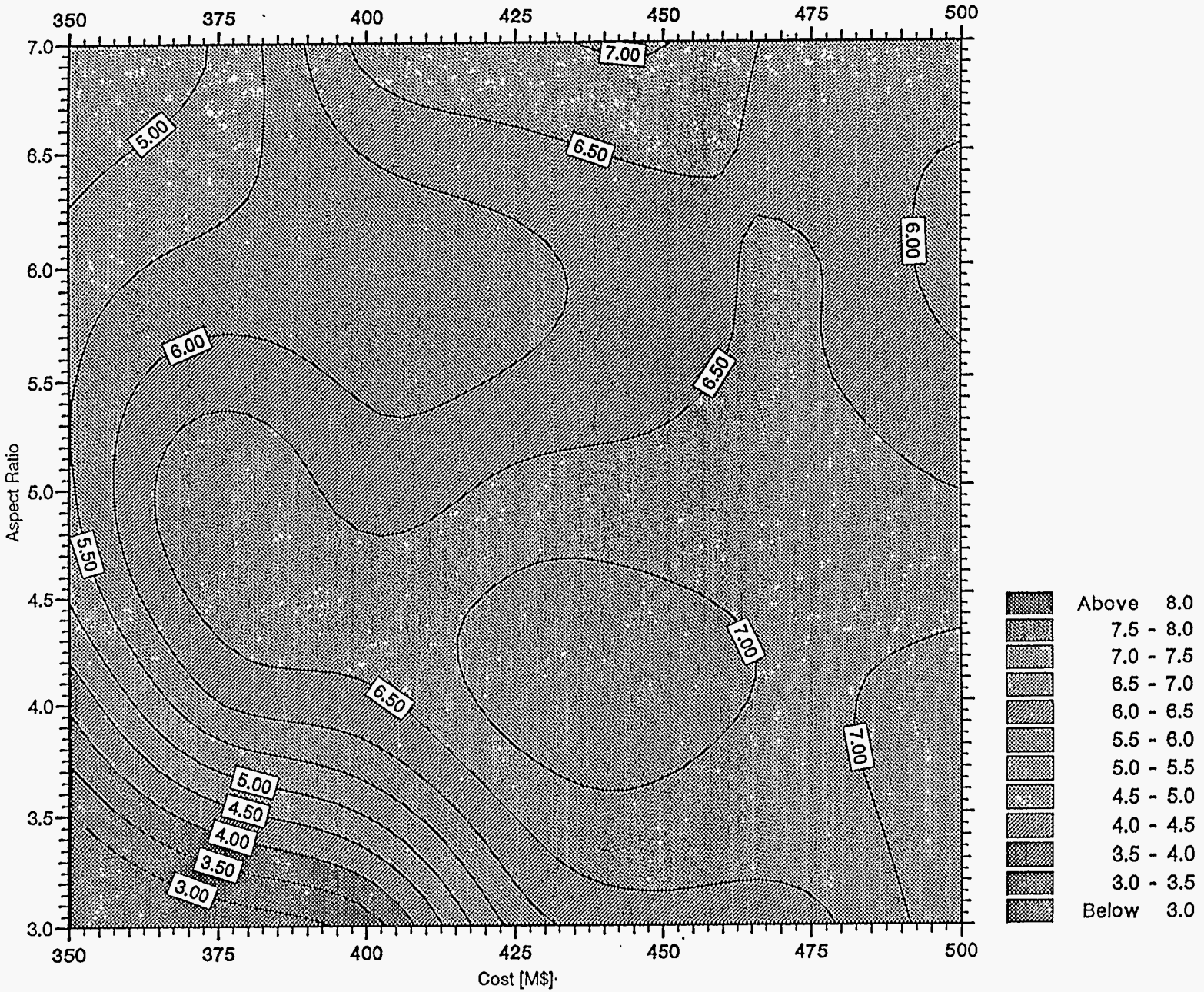
P_ICRH [MW] vs Cost and Aspect Ratio



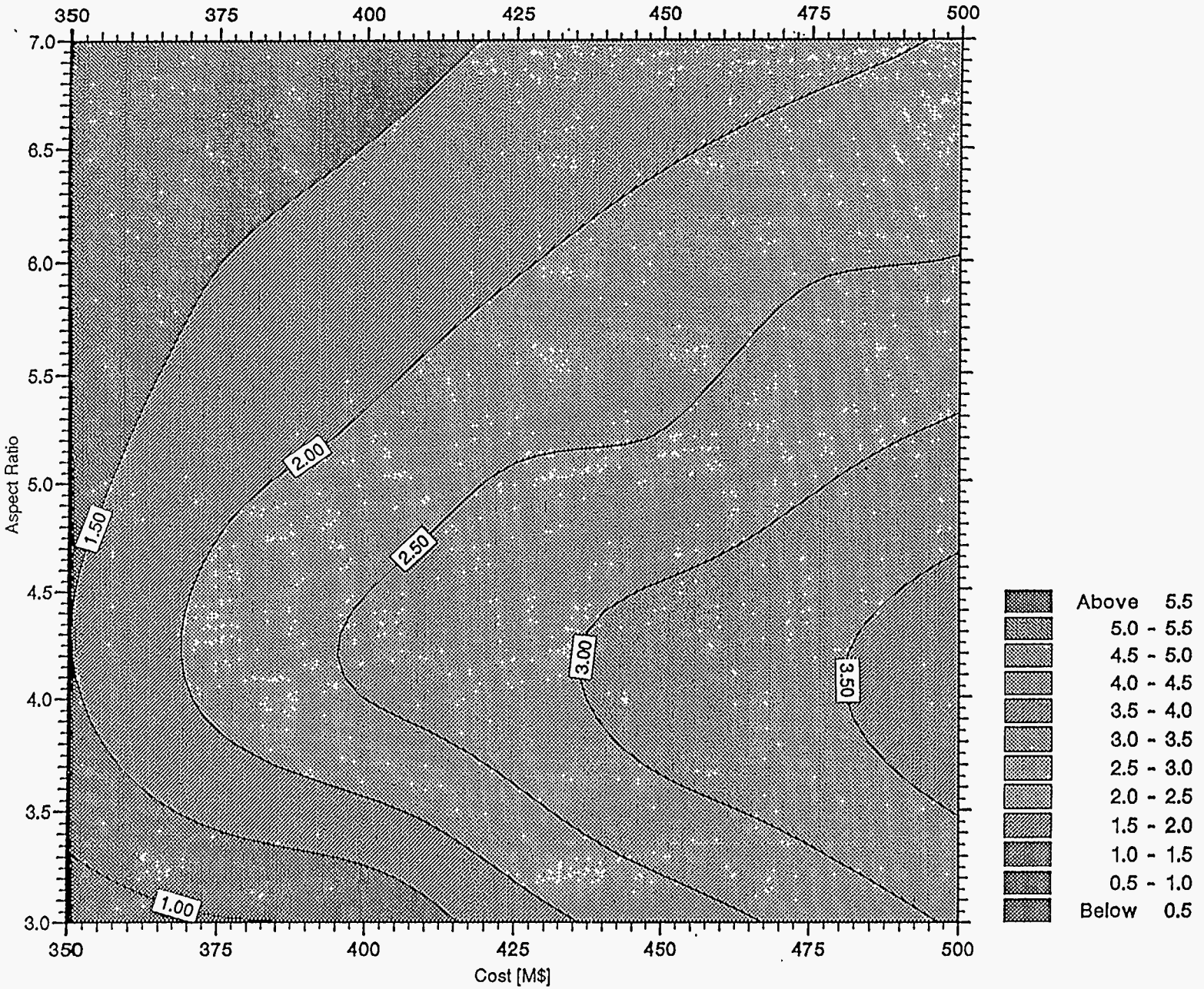
<n_e> [10^20 m^-3] vs Cost and Aspect Ratio



<T_e>_n [keV] vs Cost and Aspect Ratio



<T_I>_n [keV] vs Cost and Aspect Ratio



GRD H_dlv [MW/m] vs Cost and Aspect Ratio

SSAT DESIGN POINTS FROM A-$ SPACE

	SSAT Baseline	Machine at: $400M, A=5.0	Machine at: $350M, A=6.0
Cost (M$)	455	400	350
R (m)	2.25	2.04	1.85
A	4.5	5.0	6.0
I (MA)	1.75	1.43	0.841
B (T)	3.35	3.74	3.57
$P_{NB,\ installed}$ (MW)	16.0	24.0	16.0
$P_{IC,\ installed}$ (MW)	12.0	8.0	8.0
ST Scenario: β_N=3.5%@q_{95}=3			
β_N(%)	3.14 (Max)	3.5*	3.5*
q_0 / q_{95}	1.05*/3.0*	1.05*/3.0*	1.05*/3.0*
$n_i(0).\tau_E.T_i(0)$ (10^{20}.keV.s.m^{-3})	1.30	1.33	0.654
n_e (10^{20}m^{-3})	0.481*	0.902*	0.752*
T_e / T_i (keV)	7.17/9.21	4.78/6.25	3.72/5.19
Peak div. heat load (MW/m^2)	17.8	20.3	13.5
Div temp (eV)	354	190	136
$P_{div,\ Neilson}$ (MW/m)	2.0	2.12	1 17
AT Scenario: Max. f_{BS}@β_N=3.0%			
Bootstrap fraction	0.639(max)	0.590(max)	0.700(max)
β_N(%)	3.0*	3.0*	3.0*
q_0 / q_{95}	1.30*/7.90	1.30*/7.76	1.30*/7.36
n_e (10^{20}m^{-3})	0.30*	0.398	0.412
T_e / T_i (keV)	5.10/4.13	3.95/3.65	3.30/2.12
$P_{NB,\ used}$ (MW)	5.19	5.71	1.77
$P_{IC,\ used}$ (MW)	12.0*	8.0*	8.0*
Peak div. heat load (MW/m^2)	7.23	7.21	7.16
Div temp (eV)	551	404	276
$P_{div,\ Neilson}$ (MW/m)	1.22	1.07	0.840

* – Variable at a constraint bound

TF Coil Benchmark and Optimization for the 3.35T Baseline

	Bounds	SSAT 3.35 T Benchmark -1	SSAT 3.35 T Benchmark -2	Min Cost case
Constraints:		fraction of limit:	fraction of limit:	fraction of limit:
$\sigma_{conduit}$	$\leq$ 550 MPa	0.37	.56	1. *
σ_{case}	$\leq$ 550 MPa	0.43	.85 (.84)	1. *
V_{dump}	$\leq$ 20kV	0.20	.50 (.50)	1. *
ΔT during quench	$\leq$150 K	0.89	.98 (1.0)	1. *
Temp. headroom	$\geq$ 2K	.29	.35 (.36)	0.56
I_{op} / I_{crit}	$\leq$ 0.6	.18	.59 (.57)	1. *
I_{op} / $I_{transition}$	$\leq$ 0.8		1.15 (1.14)	1. *
Corresponding Variables:				
Δ coil (m)	$\geq$ 0.15	.3877	.228	.171
Δ case (m)	$\geq$ 0.03	.0762	.0635	.051
Δ conduit (mm)	$\geq$ 0.1	2.2	2.4	0.7
strand diameter (mm)	$\geq$ 0.1		0.78	0.51
I_{op} (kA)	$\leq$ 60	33.6	33.6	60.0 *
composite Cu fraction		.667	0.67	0.79
Cu strand fraction		0	0.33	0.21
t_{dump} (s)		10	5.5	1.4
Overall				
Coil Mass (1000 kg)			96 (97)	69
$J_{overall}$ (kA/cm^2)		1.9	2.3	3.0
Cost (M$)		61	52 (51)	46

() - numbers in parenthesis are from J. Schultz's spreadsheet.
* - at a limit

SSAT 3.35T BASELINE: MAXIMUM BOOTSTRAP FRACTION¶ SUBJECT TO DIVERTOR HEAT LOAD CONSTRAINTS

	PEAK DIVERTOR HEAT LOAD (MW/m^2) (No peaking or safety factors)			
	Unconstrained¶	≤15MW/m^2	≤10MW/m^2	≤5MW/m^2
Max bootstrap fraction	0.680	0.644	0.532	0.393
Peak div. heat load (MW/m^2)	16.7	15.0*	10.0*	5.0*
Div. temp. (eV)	608	582	452	285
T_e (keV)	6.00	5.97	4.96	3.56
T_i (keV)	7.11	7.00	6.21	4.77
n_e (10^{20} m^{-3})	0.382	0.361	0.344	0.323
P_{NB} (MW)	12.5	11.1	9.17	7.04
P_{IC}(MW)	12.0*	11.0	6.58	2.02
f_{IC}	-0.594	-0.519	-0.465	+0.863
I (MA)	1.08	1.07	1.05	1.01
q_{95}	5.11	5.11	5.12	5.14
β_N	3.50*	3.30	2.72	1.99
β_{fast}/β	0.25*	0.25*	0.25*	0.25*

¶ Earlier, preliminary optimized case. See update at front of package

SSAT 3.35T BASELINE: MAXIMUM¶ β_N at q95=3 (I=1.67MA), SUBJECT TO DIVERTOR HEAT LOAD CONSTRAINTS

	PEAK DIVERTOR HEAT LOAD (MW/m^2) (No peaking or safety factors)					
	Unconstrained¶	≤20MW/m²	≤15MW/m²	≤10MW/m²	≤8MW/m²	≤7.5MW/m²
max β_N	2.95	2.90	2.48	1.83	1.31	No solution
Peak div. heat load (MW/m^2)	20.6	20.0*	15.0*	10.0*	8.0*	—
Div. temp. (eV)	519	505	421	351	362	—
T_e (keV)	8.08	7.93	6.76	6.11	6.74	—
T_i (keV)	10.3	10.2	9.28	7.76	6.44	—
n_e (10^{20} m^{-3})	0.372	0.371	0.356	0.306	0.234	—
P_{NB} (MW)	12.7	12.5	11.1	7.21	3.69	—
P_{IC}(MW)	12.0*	11.5	7.71	5.87	6.65	—
f_{IC}	-0.039	-0.019	+0.027	+1.0*	+1.0*	—
Bootstrap fraction	0.407	0.399	0.355	0.250	0.187	—

¶ Earlier, preliminary optimized case. See update at front of package.

SSAT 3.35T BASELINE: MAXIMUM BOOTSTRAP FRACTION¶ SUBJECT TO DIVERTOR TEMPERATURE CONSTRAINTS

	DIVERTOR TEMPERATURE (eV)			
	Unconstrained¶	**≤300eV**	**≤220eV**	**≤200eV**
Max bootstrap fraction	0.680	0.569	0.550	No solution
Div. temp. (eV)	608	300*	220*	—
Peak div. heat load (MW/m^2)	16.7	19.3	18.2	—
T_e (keV)	6.00	4.80	4.24	—
T_i (keV)	7.11	5.18	4.36	—
n_e (10^{20} m^{-3})	0.382	0.730	0.890	—
P_{NB} (MW)	12.5	16*	16*	—
P_{IC}(MW)	12.0*	12.0*	12.0*	—
f_{IC}	-0.594	+0.135	+0.810	—
I (MA)	1.08	1.51	1.60	—
q_{95}	5.11	3.52	3.31	—
β_N	3.50*	3.14	3.02	—

¶ Earlier, preliminary optimized case. See update at front of package

SSAT TF COIL OUTER LEG TRADEOFF STUDY

- A tradeoff study has been performed with the SUPERCODE, to assess the effect of the NB tangency radius on the radius of the outer leg of the TF coil and therefore on the cost of the machine.

- As the tangency radius moves to the outer part of the plasma, the NB current drive efficiency becomes larger. At the same time the required size of the port for NB access increases, driving up the radius of the outer leg of the TF coil.

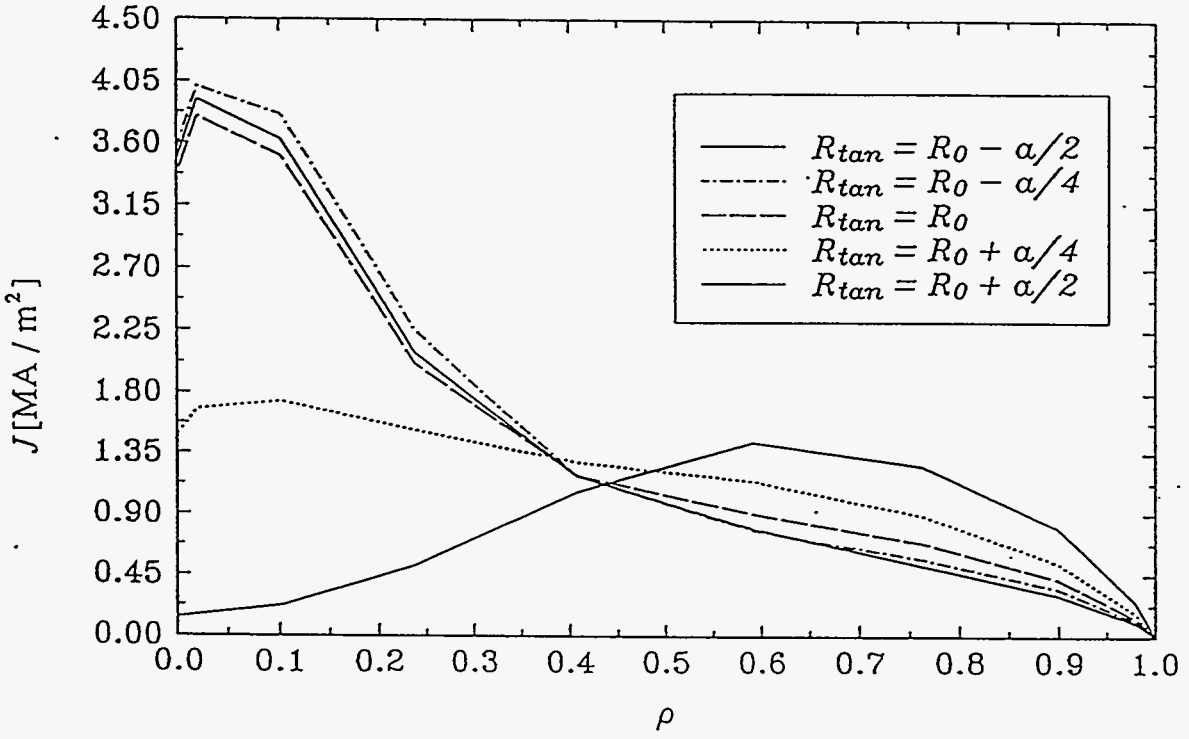
- A series of simulations has been performed for the fixed-size SSAT baseline for different positions of the NB tangency radius. The NB current drive efficiency and the NB current density and heating profiles are calculated self-consistently using SUPERCODE's profile-dependent neutral beam module. These runs try to maximize β, under the constraint that the required port size for beam access is $\leq$ than the available port size. The size of the port between the TF coils is adjusted by varying the size of the gap between the outboard shield and the TF Coil dewar. The results are summarized in the table below:

R_{tan}	R_0-$a/2$	R_0-$a/4$	R_0	R_0+$a/4$	R_0+$a/2$
Relative Cost	1.00	1.004	1.010	1.015	1.022
TF Coil Weight (kg)	8844.5	8951.07	9086.04	9224.76	9377.62
TF Leg Gap (m)	0.1042	0.1942	0.3053	0.4168	0.5368
Required Port Size, (m)	1.0345	1.0560	1.0746	1.0918	1.1078
β	0.03204	0.03206	0.03211	0.3222	0.03237
NB Current (MA)	1.0313	1.096	1.180	1.295	1.440
ICRH Current (MA)	-0.119	-0.183	-0.266	-0.378	-0.520
NB efficiency η (A/W)	0.0644	0.0685	0.0738	0.080	0.090
TF coil ripple (%) (peak-to-average)	0.1569	0.1111	0.07327	0.0487	0.0318

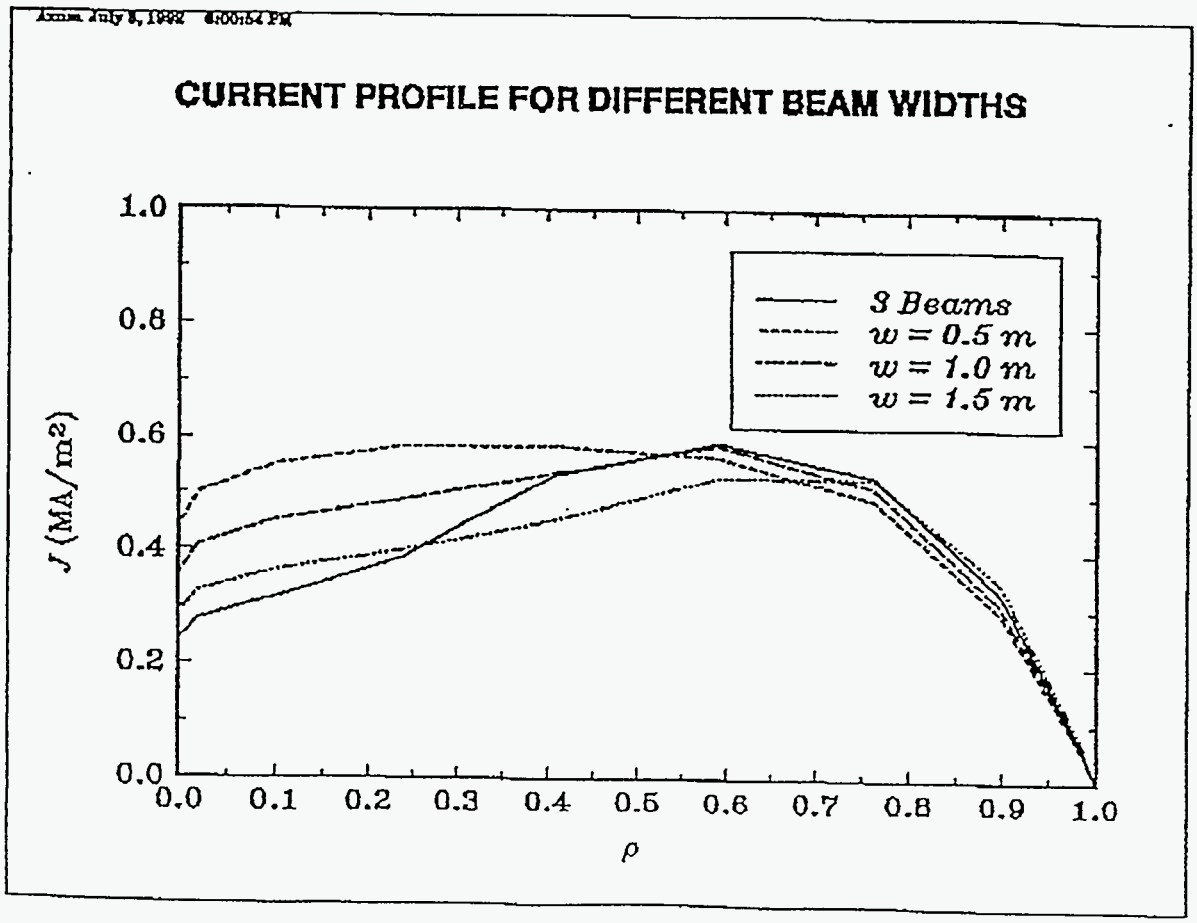
- Results indicate that cost is not very sensitive to the position of the NB tangency radius. The position giving us the best plasma performance, should be selected.

- The MHD equilibrium in the above calculations, is not consistent with the changing NB-driven current profile during the R_{tan} scan. This is an important effect and will be addressed in the next version of the code.

SSAT TF COIL OUTER LEG TRADEOFF STUDY



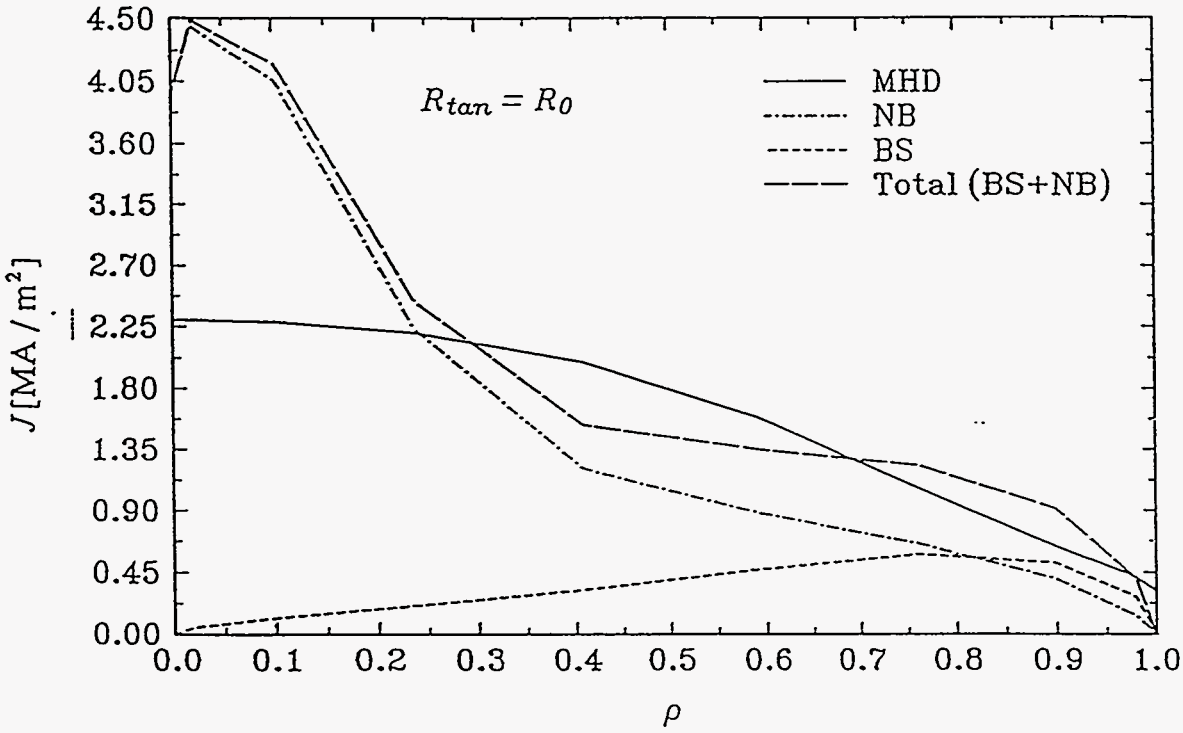
Assumption: 1 beam of 0.5m width at plasma



— = 3 beams (3 sources) with cross over at duct center (0.5 m width) per TFTR/SSAT specs. $R_{tan} = R_o - a/2$

1-1/2-D Effects

- We compute 2-D equilibria based on partial transport information:
  - $p'(\psi)$ computed consistent with transport.
  - $FF'(\psi)$ computed using parametric J_ϕ profile.
- Code will soon eliminate this inconsistency.



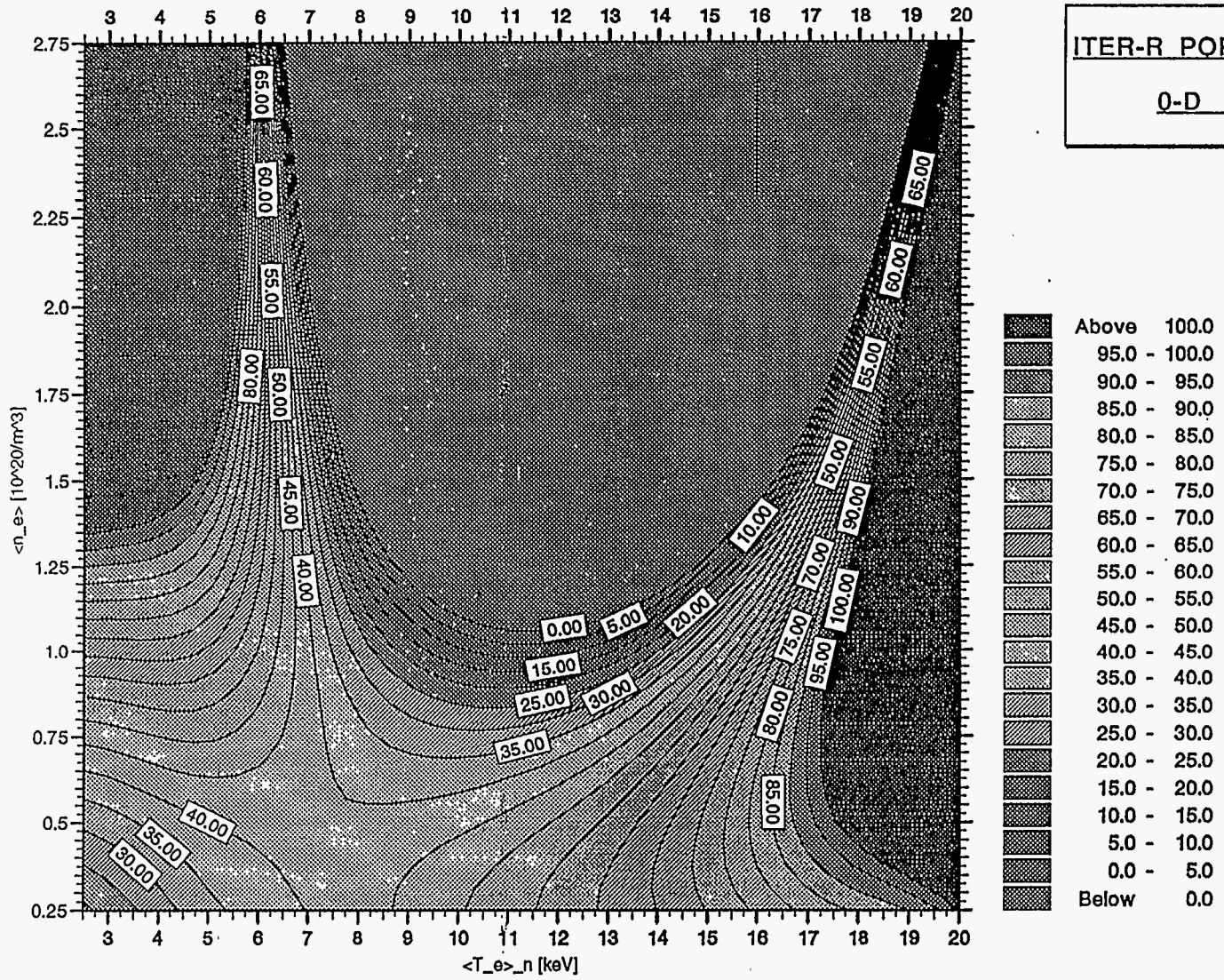
JT-60 Super-Upgrade

Parameter	High Current	SS + High B.S. Fraction	SS + High $n\tau T$
R_0 [m]	4.3	4.3	4.3
a [m]	1.3	1.3	1.3
I_p [MA]	10	3.52	5.92
B_0 [T]	5.2	5.2	5.2
$q_\psi(0)$	1.05	1.05	1.05
$q_\psi(95\%)$	2.97	8.4	5.0
I_{SS}/I_p	0.42	1.0	1.0
I_{BS}/I_p	0.17	0.47	0.28
β_N	0.012	0.018	0.015
β	0.018	0.009	0.013
$\beta_p(1)$	0.49	2.1	1.04
$\langle n_e \rangle$ [m^{-3}]	0.71	0.35	0.44
$\langle T_e \rangle_n$ [keV]	8.31	7.65	9.08
$\langle T_i \rangle_n$ [keV]	8.08	7.07	8.72
P_{NB} [MW]	30	30	30
P_{ICRH} [MW]	30	30	30
τ_E [sec]	1.29	0.53	0.834
$n\tau T$ [10^{20} keV−sec/ m^3]	16.7	2.85	7.16
H_{div} [MW/ m^2]	5.96	5.78	6.65
T_{div} [eV]	94	569	324
Cost [M$]	1252	←	←

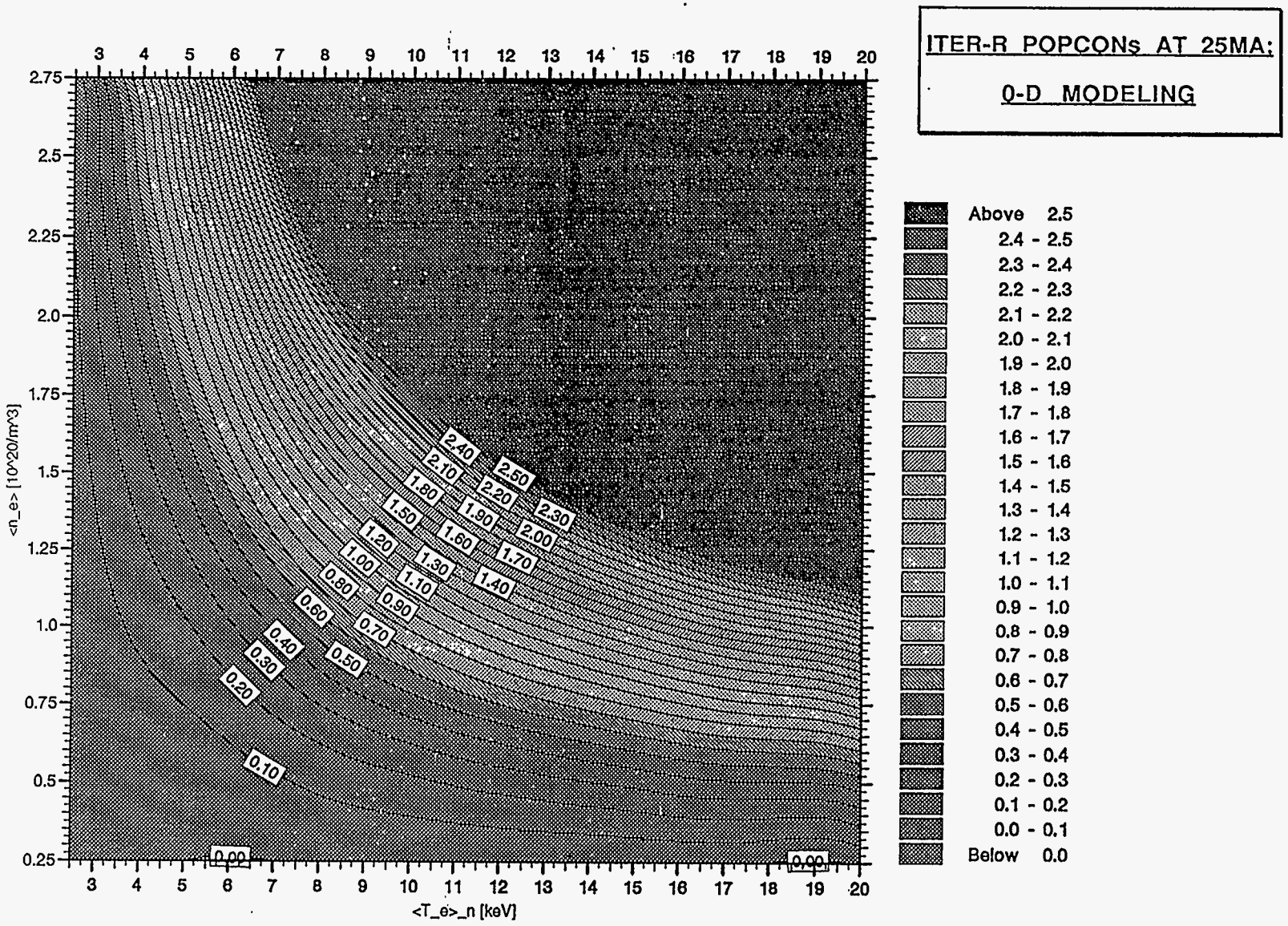
ITER-R (R=8M) AT 1MW/m^2 NEUTRON WALL LOAD: 0-D -v- 1-D MODELING OF IGNITION PERFORMANCE

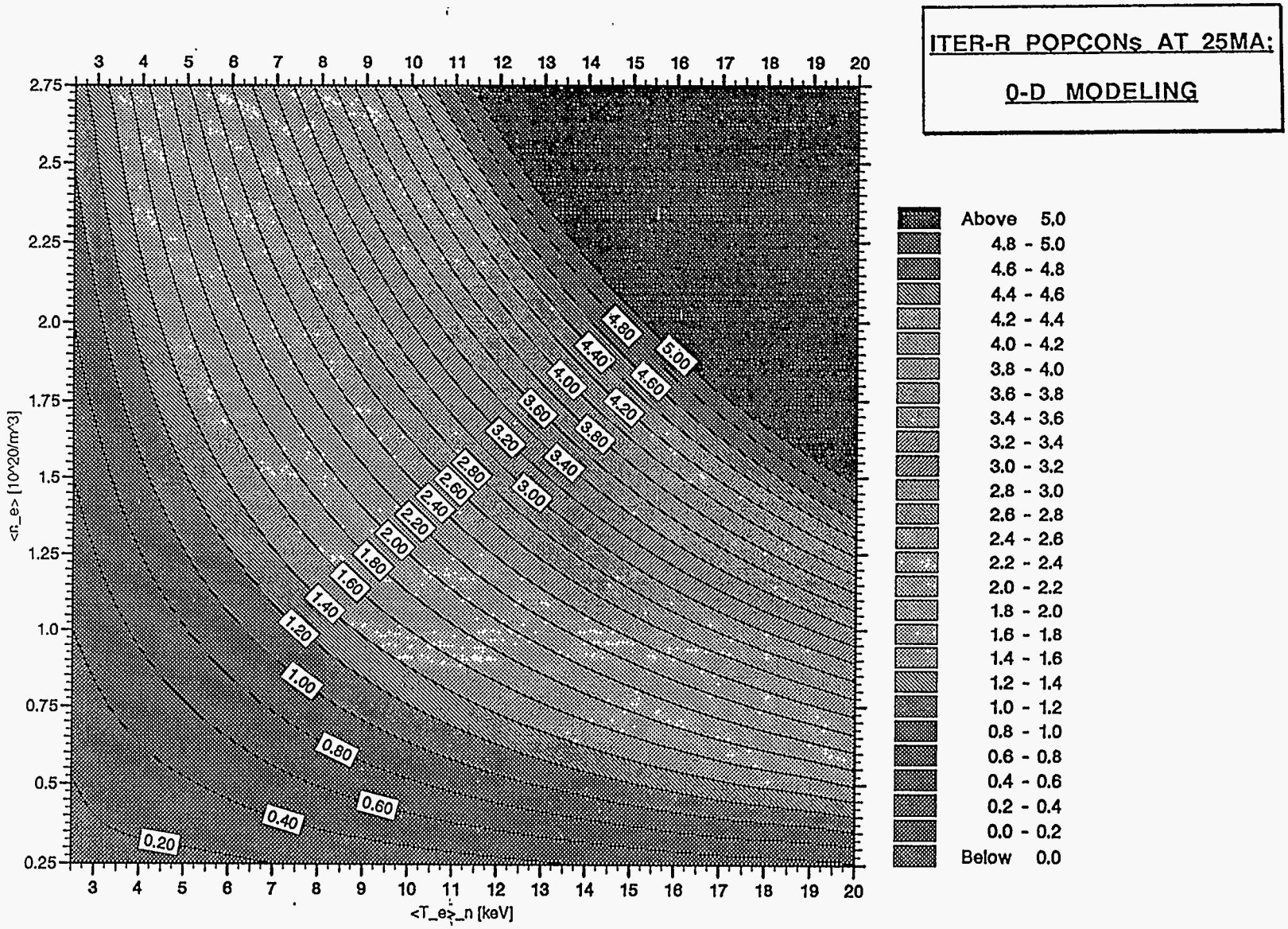
	0-D PHYSICS UNDER CDA "RULES"	1-1/2D PHYSICS WITH Rebut ∇T_c χ's
Conditions	Ignited under 0-D	Ignited under 1-D ∇T_c
Plasma current (MA) / q_ψ(95%)	25 / 3.14	←
Average neutron wall load (MW/m^2)	1.0*	←
Fusion power (MW)	1730*	←
$\langle T_e \rangle_n$ (keV)	10.0*	12.1
$T_e(0)$ (keV)	(17.3)	22.1
Temperature peaking T(0) / <T>	2.0*	2.11
$\langle n_e \rangle$ (10^{20} m^{-3})	1.10	1.20
Density peaking $n_e(0) / \langle n_e \rangle$	1.5*	1.26
Impurities:		
$f_{He} = n_{He} / n_e$ (%)	10*	20.0
$f_{Be} = n_{Be} / n_e$ (%)	—	1.0*
$f_C = n_C / n_e$ (%)	1.08(b)	—
$f_O = n_O / n_e$ (%)	0.100(b)	—
$f_{Fe} = n_{Fe} / n_e$ (%)	0.018(b)	—
Z_{eff}	1.70	1.52
Plasma stored energy (MJ)	1060	1329
P_{rad} (MW)	124	113
Troyon beta coefficient(e) (%)	1.76	2.22
Req'd τ_E (s)	4.76	5.68
Req'd $n_{DT}(0).\tau_E.T_i(0)$ (10^{20} m^{-3}.s.keV)	98.0	103
Req'd 0-D enhancement factors:		
ITER-P scaling	1.93	2.34
ITER-H scaling(c)	0.726	0.880
Rebut-Lallia scaling(a)	1.61	1.85

* Fixed input requirements or constraint bound. (a) Original (1989) R-L 0-D offset-linear 0-D scaling.
(b) CDA impurity models =f(n_e). (c) New ITER H-mode scaling due to K.Reidel (Nuclear Fusion, July '92).



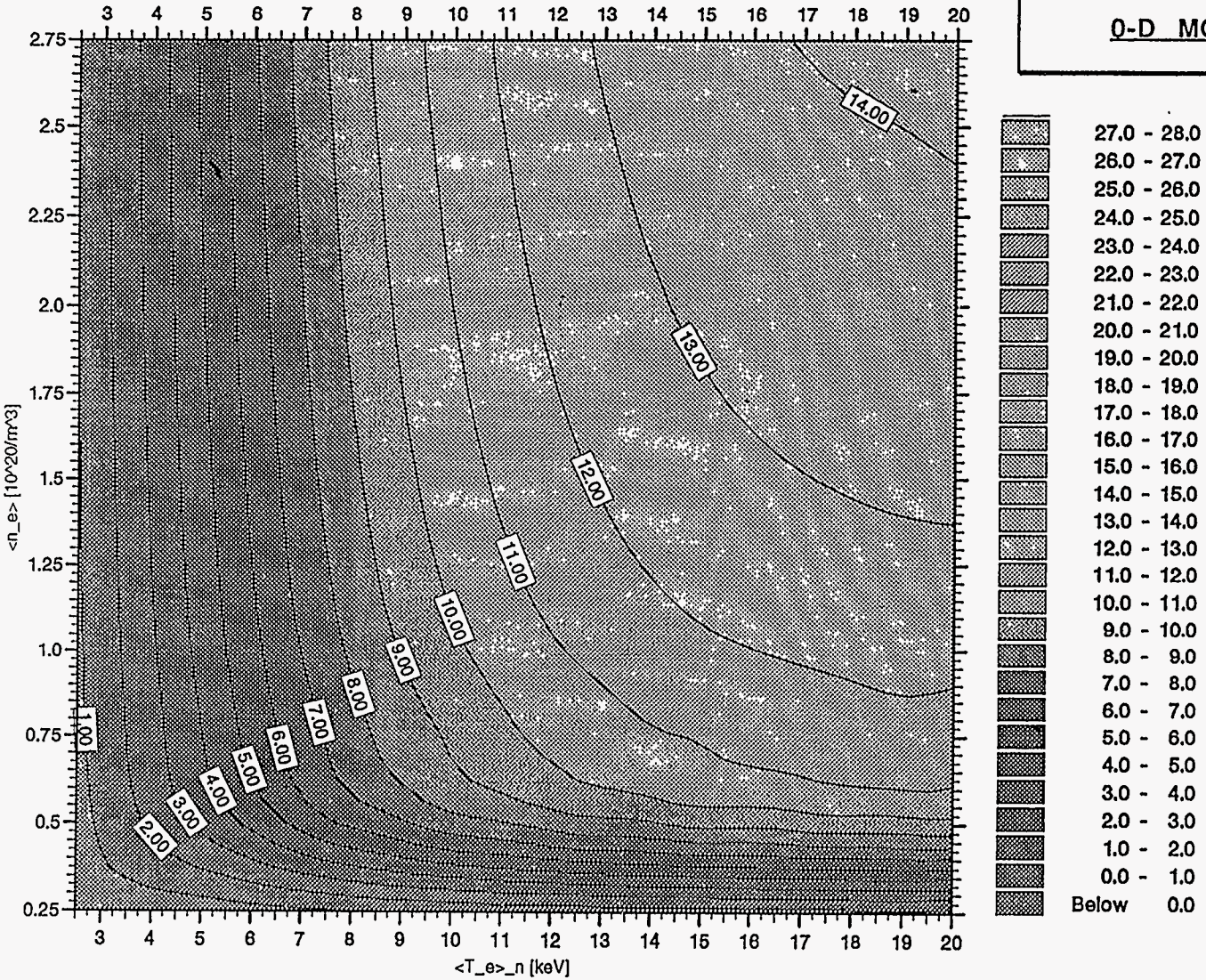
Auxillary power contours [MW]


ITER-R POPCONs AT 25MA:
0-D MODELING
Above 2.5
2.4 - 2.5
2.3 - 2.4
2.2 - 2.3
2.1 - 2.2
2.0 - 2.1
1.9 - 2.0
1.8 - 1.9
1.7 - 1.8
1.6 - 1.7
1.5 - 1.6
1.4 - 1.5
1.3 - 1.4
1.2 - 1.3
1.1 - 1.2
1.0 - 1.1
0.9 - 1.0
0.8 - 0.9
0.7 - 0.8
0.6 - 0.7
0.5 - 0.6
0.4 - 0.5
0.3 - 0.4
0.2 - 0.3
0.1 - 0.2
0.0 - 0.1
Below 0.0
<n_e> [10^20/m^3]
<T_e>_n [keV]
Neutron wall loading contours [MW/m^2]


ITER-R POPCONs AT 25MA;
0-D MODELING
Above 5.0
4.8 - 5.0
4.6 - 4.8
4.4 - 4.6
4.2 - 4.4
4.0 - 4.2
3.8 - 4.0
3.6 - 3.8
3.4 - 3.6
3.2 - 3.4
3.0 - 3.2
2.8 - 3.0
2.6 - 2.8
2.4 - 2.6
2.2 - 2.4
2.0 - 2.2
1.8 - 2.0
1.6 - 1.8
1.4 - 1.6
1.2 - 1.4
1.0 - 1.2
0.8 - 1.0
0.6 - 0.8
0.4 - 0.6
0.2 - 0.4
0.0 - 0.2
Below 0.0
<n_e> [10^20/m^3]
<T_e>_n [keV]
Troyon beta coefficient contours [%]

ITER-R POPCONs AT 25MA:

0-D MODELING



Alpha ash fraction contours [%]

Commercial Reactor Extrapolations: Rebut ITER-R Philosophy -v- TPX Philosophy

- ***Find the minimum unit size commercial reactor subject to the following rules:***

	"ITER-R-like" Reactor	"TPX-like" Reactor
Input Rules:		
Maximum confinement, H	*≤2*	*≤4*
Maximum Troyon Coeff. β_N	*≤3*	*≤5*
Minimum Q	∞ (ignited)	*≥15*
Minimum burn time	*10 hours inductive*	*Steady-State (10 months[a])*
Aspect ratio	*3*	*4.5*
Output results:		
Minimum Unit Size [MW_e]	2750(!)	750
Cost [B$]	24	6.5
COE [mills / kW−hr]	157	160
Unit Cost [$ / kW_e]	8730	8670
R_0 [m]	12	6.3
I_p [MA]	21	9.8
B_0 [T]	6.2	7.1
Q	∞	16
H	2*	2.3
β_N [%]	3*	5
β_t [%]	2.5	5
Burn pulse length	10 hours*	Steady-State (10 months[a])

* -- variable at a constraint bound or fixed. a -- On-time between scheduled maintenance

WHAT DOES HIGHER ELONGATION BUY YOU?

Rules for this Study:

Vary 95% plasma elongation in the range 1.6 to 2.2 and, for each elongation, find the minimum size/cost, Rebut-like machine, under the following constraints:

- ***Ignited under Rebut $(\nabla T_e)_{crit}$ transport at 1MW/m^2 neutron wall load (NB: 0-D H factor not held constant)***
- ***Same plasma current as the ITER-R baseline(25MA)***
- ***Low aspect ratio regime, $2.6 \leq A \leq 3.0$***
- ***Burntime $\geq$ that of the ITER-R baseline***
- ***$q_{95} \geq$ that of the ITER-R baseline***
- ***Same engineering allowables (stresses, builds, etc) as ITER-R***

	ITER-R BASELINE (k_{95} = 1.58)	MINIMUM COST, REBUT-LIKE MACHINES			
		k_{95} = 1.60	k_{95} = 1.80	k_{95} = 2.00	k_{95} = 2.20
Configuration					
Construction cost (B$)	5.80 (1.0)	5.34 (0.92)	4.96 (0.85)	4.83 (0.83)	4.73 (0.81)
Major radius (m)	8.0	7.96	7.53	7.22	6.96
Minor radius (m)	3.0	3.06	2.90	2.78	2.68
Aspect ratio	2.67	2.60*	2.60*	2.60*	2.60*
Axial field (T)	5.81	5.50	4.86	4.66	4.51
TF stored energy (GJ)	108	93.0	67.5	60.3	55.5
TF coil mass (tonnes)	11,100	8,644	6,663	6,132	5,659
Ignition performance:					
Plasma current (MA)	25.0*	25.0*	25.0*	25.0*	25.0*
q_ψ(95%)	3.14	3.17	3.30	3.58	3.93
Av. neut. wall load (MWm^{-2})	1.0*	1.0*	1.0*	1.0*	1.0*
Fusion power (MW)	1730	1760[a]	1730[a]	1710[a]	1720[a]
Equiv H fact –ITER-P scaling	2.34[b]	2.12	1.90	1.84	1 80
Burn pulse length (s)	1380	1380*	1380*	1380*	1380*

** Variable fixed or at a constraint bound a -- Fusion power remains ~constant as size decreases due to increasing k_{95}. b -- 0-D H factor not constrained here as 1-D transport applied.*

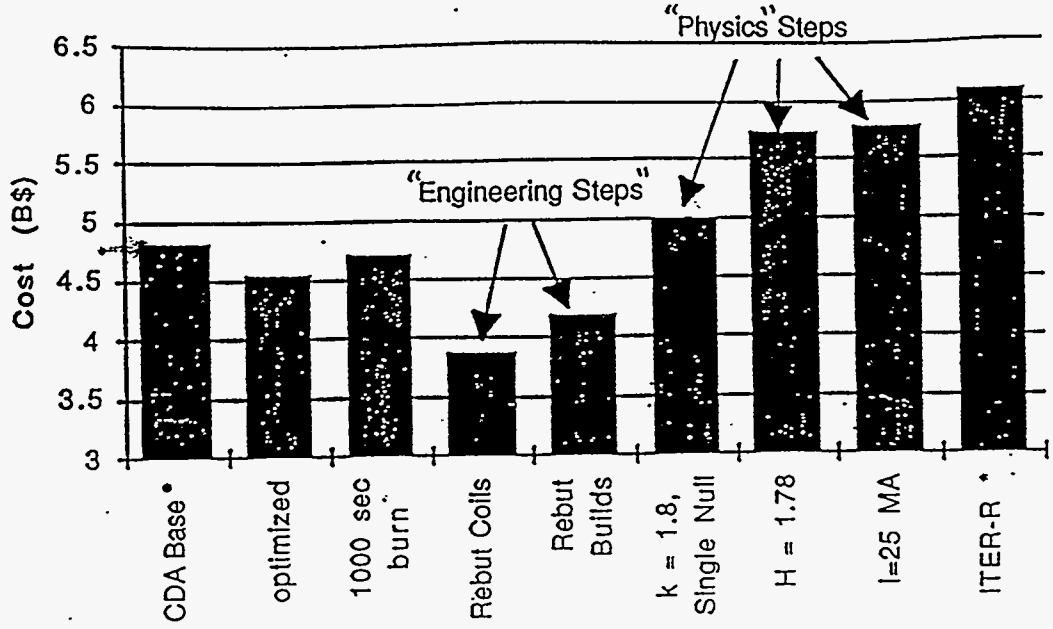


Fig 1. TRANSITION FROM THE CDA TO ITER-R; OPTIMUM ASPECT RATIO

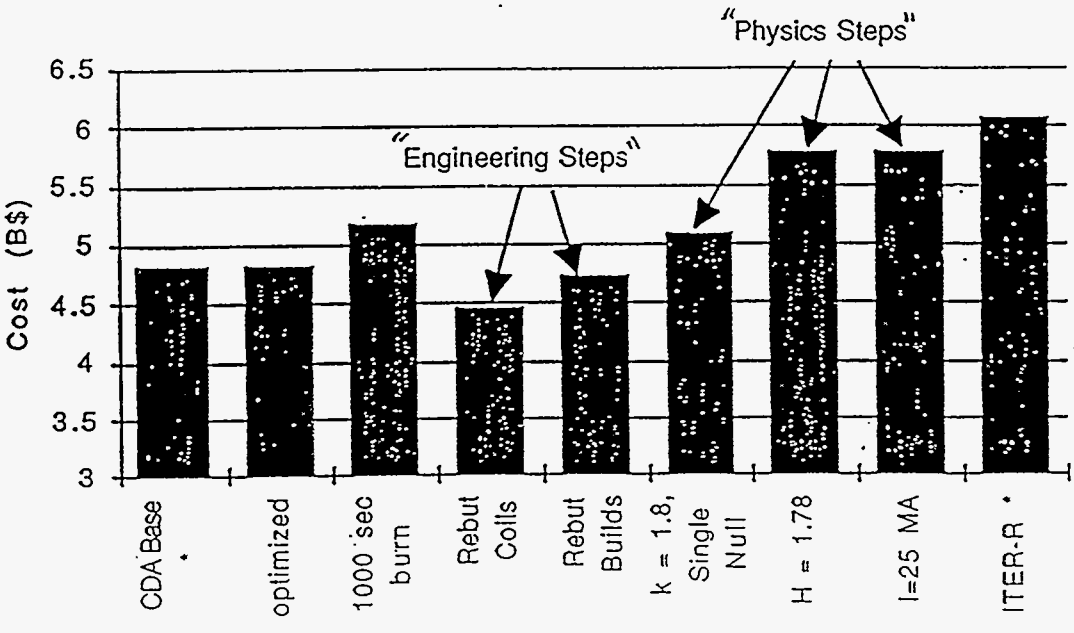


Fig 2. TRANSITION FROM THE CDA TO ITER-R; FIXED ASPECT RATIO OF 2.8

Table 5. HOW WE GET FROM THE CDA TO ITER-R

	CDA *	CDA min. cost	1000 s burn	ITER-R coils	ITER-R builds	Shape: SN, κ=1.66	Confine-ment: H_{ITER-P} = 1.78	$I_p \geq 25$ MA	ITER-R*
Cost (B$)	4.82	4.54	4.72	3.87	4.19	4.96	5.72	5.78	6.08
% change in cost	0	-6%	-2%	-22%	+8%	+18%	+15%	+1%	+5%
R (m)	6	5.87	6.17	5.63	5.96	7.08	7.82	7.72	7.74
A	2.8	3.4	3.5	3.8	3.8	3.1	2.9	2.8	2.8
B_0 (T)	4.9	6.1	6.2	7.2	6.7	6.2	6.2	6.0	6.0
I_p (MA)	22	17.5	16.9	14.9	15.1	19.7	23.9	25	25
β (%)	3.8	2.7	2.6	2.1	2.5	2.4	2.2	2.3	2.3
Troyon coeff	1.8	1.6	1.7	1.5	1.8	1.7	1.5	1.5	1.5
H_{ITER-P}	2.0	2.0	2.0	2.0	2.0	2.0	1.8	1.8	1.8
P_{fusion} (MW)	1080	870	920	720	940	1360	1700	1710	1720
B_{max-TF} (T)	11.2	12.6	12.3	13.3	13.1	12.9	12.8	12.7	12.7
W_{TF} (GJ)	41	51	55	54	70	82	107	102	107
H_{div} (MW/m^2)	7.2	6.0	5.9	5.3	6.5	8.6	8.9	9.2	9.2

* - no optimization

Table 6. HOW WE GET FROM THE CDA TO ITER-R -- FIXED ASPECT RATIO OF A=2.8

	CDA*	CDA** min. cost	1000 s burn	ITER-R coils	ITER-R builds	Shape: SN, κ=1.66	Confine-ment: H_{ITER-P} = 1.78	$I_p \geq 25$ MA	ITER-R*
Cost (B$)	4.82	4.82	5.18	4.47	4.73	5.10	5.79	<-	6.08
% change in cost	0	0	+7%	-16%	+6%	+8%	+14%	0	+5%
R (m)	6	6	6.41	5.98	6.32	6.94	7.70	<-	7.74
A	2.8	2.8	2.8	2.8	2.8	2.8	2.8	<-	2.8
B_0 (T)	4.9	4.9	4.7	4.9	4.6	5.7	6.0	<-	6.0
I_p (MA)	22	22	22.4	21.9	22.0	21.9	25.1	<-	25
β (%)	3.8	3.8	4.0	3.8	4.3	2.7	2.3	<-	2.3
Troyon coeff	1.8	1.8	1.9	1.8	2.1	1.8	1.5	<-	1.5
H_{ITER-P}	2.0	2.0	2.0	2.0	2.0	2.0	1.8	<-	1.8
P_{fusion} (MW)	1080	1080	1220	1070	1330	1410	1710	<-	1720
B_{max-TF} (T)	11.2	11.2	10.6	10.6.	10.7	12.8	12.8	<-	12.7
W_{TF} (GJ)	41	41	43	38	49	75	102	<-	107
H_{div} (MW/m^2)	7.2	7.2	7.3	7.1	8.3	9.1	9.2	<-	9.2

* - no optimization. ** Same as with no optimization

ITER-R (R=7.75M, I=25MA): IGNITION CHARACTERISTICS VERSUS NEUTRON WALL LOAD / FUSION POWER

	Average Neutron Wall Load (MW/m^2)					
	0.5	1.0	1.5	2.0	2.5	3.0[f]
Peak neutron wall load (MW/m^2)[e]	0.639	1.28	1.93	2.59	3.25	3.91
Fusion power (MW)	861	1720	2580	3440	4310	5170
Thermal power (MW)[a]	~1170 – 1310	~2340 –2620	~3510 –3930	~4680 – 5240	~5860 – 6791	~7030 – 7860
Total construction cost (B$)[b]	5.92	6.09	6.24	6.40	6.55	6.69
$\langle n_e \rangle$ (10^{20} m^{-3})	0.822	1.13	1.38	1.59	1.77	1.94
Troyon beta coefficient (%)	1.10	1.54	1.87	2.16	2.41	2.64[f]
Req'd τ_E (s)	7.30	4.19	3.20	2.68	2.34	2.11
Req'd $n_{DT}(0).\tau_E.T_i(0)$ (10^{21} m^{-3}.s.keV)	105	85.2	79.6	76.9	75.3	74.2
Req'd enhancement factors for ignition:						
ITER-P	2.05	1.78	1.68	1.63	1.59	1.56
ITER-H[c]	0.787	0.676	0.636	0.614	0.599	0.588
Rebut-Lallia[d]	2.32	1.56	1.25	1.07	0.954	0.868
Burn pulse (s)[g]	850	1070	1220	1350	1460	1560
Peak divertor heat flux (MW/m^2)[h]	4.00	9.15	14.8	20.7	26.8	33.1

(a) Lower end of range for Li-bearing blankets; upper end of range for steel shielding "blankets"

(b)Cost increases with fusion power due to increases in reactor cooling, heat exchangers, PF system, fueling and heat rejection. Note that factor of 3 in fusion power costs an additional ~600M$ (~10%).

(c) New ITER H-mode scaling due to K.Reidel (NF July '92).

(d) Original (1990) RL 0-D scaling. See later for 1-D treatment with Rebut-Lallia -Watkins χ's

(e) Peak neutron wall loading at outboard midplane (~ front face center of test module).

(f) Note Troyon beta coeff of 2.64% is req'd for av. wall loading of $3MW/m^2$ (5170MW fusion power). A Troyon limit of 3% would permit average wall loads close to $4MW/m^2$ (~6900MW fusion power).

(g) Burn pulse length increases with fusion power due to higher beta-p ($\rightarrow$ higher BS fraction and greater V.s contribution from outer vertical field coils)

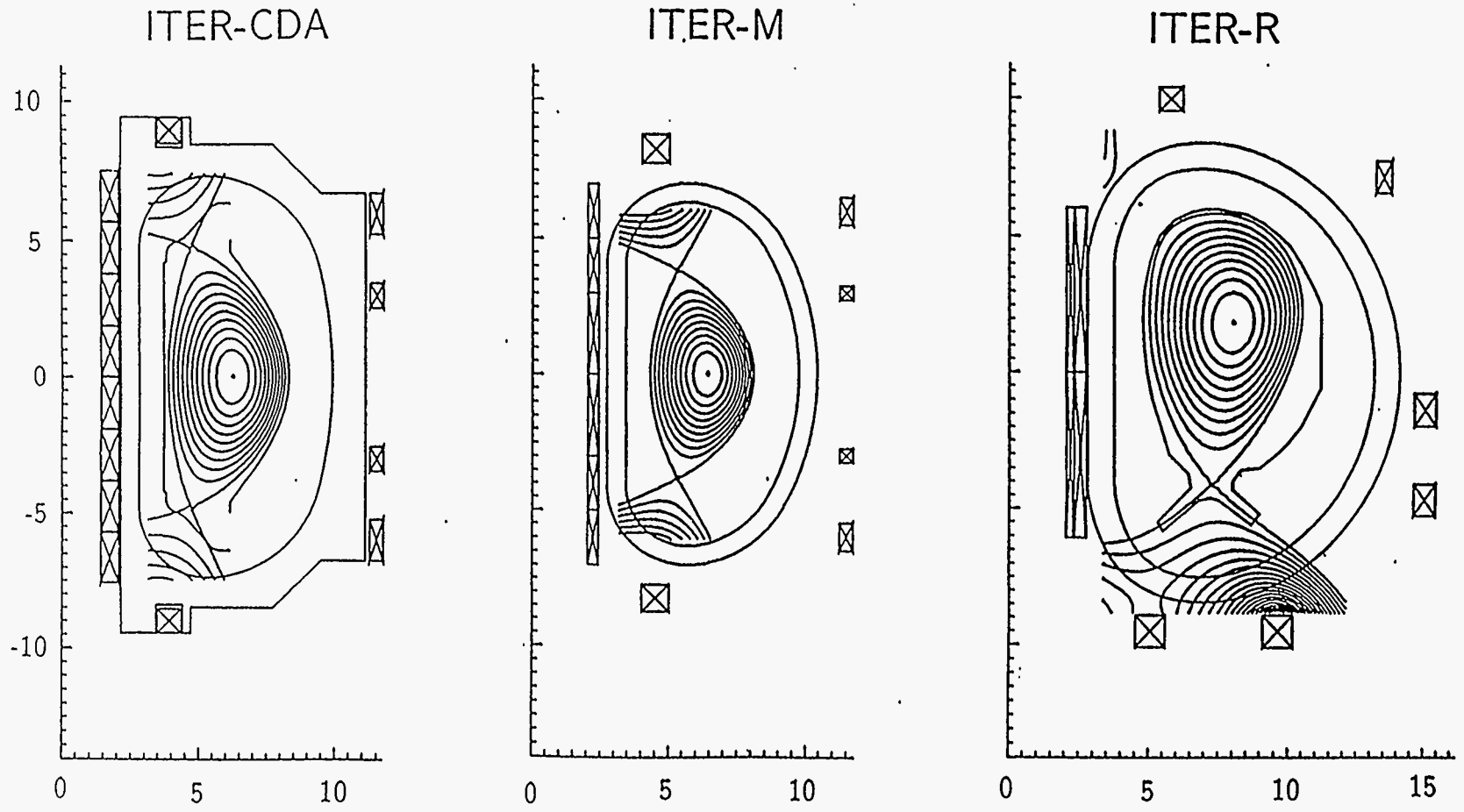


Fig. 1. Comparison of ITER-M with the CDA and the Rebut ITER Proposal, ITER-R

Table 1. Comparison of ITER-M, the CDA , and ITER-R: Design Parameters and Ignition Performance*.

	ITER-M	ITER CDA	REBUT CANDIDATE DESIGN, ITER-R
Configuration			
Major radius (m)	6.19	6.0	7.75
Minor radius (m)	1.89	2.15	2.75
Aspect ratio	3.28	2.79	2.81
Plasma current (MA) [1]	16.8	22.0	25.0
Axial field (T)	6.52	4.85	6.0
Elongation (95%)	1.8 (DN)	2.0 (DN)	1.66 av. (SN)
q_ψ(95%)	3.0	3.0	2.87
Plasma thermal energy (GJ)	0.535	0.564	0.967
Plasma inductive energy (GJ)	1.72	2.22	4.32
Peak TF field(T) [2]	12.8	11.3	12.7
No. TF coils	24	16	24
TF support	bucked thru OHC	wedged	bucked thru OHC
TF stored energy (GJ)	61.7	40.7	108
TF coil mass (tonnes) [3]	5,650	6,960	11,100
TF coil costs (B$) [4]	0.528	0.575	0.935
Construction cost (B$) [5]	4.45	4.83	6.09
Ignition performance:			
Av. neut. wall load (MW/m^2)	1.5	1.0	1.0
Pk. neut. wall load (MW/m^2) [6]	2.0	1.4	1.3
Fusion power (MW)	1350	1080	1720
Thermal power (MW) [7]	~ 1840-2060	~1470-1640	~2340-2620
Troyon beta coefficient	2.02	1.81	1.54
Ignition enhancement factor:			
ITER89 - P	1.98	1.98	1.78
ITER H-mode [8]	0.788	0.809	0.676
Rebut-Lallia [9]	2.17	2.02	1.56
Inductive burn time (s)	1000	400	1070 [10]
Peak div. heat flux (MW/m^2)	8.3	7.2	9.2

* See notes over

Notes on Table 1

[1] Plasma current for inductive ignited operation (see over for non-inductive operation).

[2] True peak TF field at the conductor with distributed winding pack

[3] Mass of TF coils only (including bucking cylinder for ITER-R and ITER-M). No intercoil structure included here.

[4] Total TF coil cost including inter-coil support. The cost per unit mass would come out as follows for the coil mass and cost only (+buck. cyl. for ITER-R and ITER-M) : 75.5$/kg for CDA 83.6 $/kg for ITER-M and 72.9$/kg for ITER-R.

[5] See later Table for full cost breakdown.

[6] Peak neutron wall loading at outboard midplane (~ front face center of test module) from 2D integration over fusion source geometry.

[7] Lower end of range for Li-bearing blankets; upper end of range for steel (shielding) blankets

[8] New ITER H-mode scaling due to K.Reidel (NF July '92).

[9] Original (1990) RL 0-D scaling. See Ref. 1 for comparison with 1-D treatment using Rebut-Lallia - Watkins transport coefficients

[10] Based on peak OH field of 12.8/12.9 T @ BOP/EOF.

Table 2. Preliminary Cost Breakdown Comparison of the ITER-M, CDA and ITER-R Devices.

	ITER-M	ITER CDA	REBUT CANDIDATE DESIGN, ITER-R
Site and Buildings	453	504	658
Reactor Vessel	628	738	824
First wall	92	108	166
Blanket	172	221	—
Shield + vac. vessel	298	344	569
Structure	20	25	48
Divertor	57	50	42
Magnets	870	1022	1600
TF coils	528	575	935
PF coils	245	340	517
cryostat	98	107	148
Injection power	446	446	446
Vacuum	70	40	46
Power Conditioning	161	136	195
Heat Transport	312	341	373
Fueling	160	162	205
I&C	150	150	150
Maintenance Equipment	125	125	125
Electric Plant, Miscellaneous, Heat Rejection Fluid Supply	166	160	214
Total Direct Cost	3540	3840	4840
Total Constructed Cost [a]	4450	4830	6090

(a) - assumes an 18% overall contingency and an 8% add on for assembly and transport.

STEADY-STATE CURRENT-DRIVE:

SCENARIOS WERE CONSTRUCTED USING THE OPTIMIZING SYSTEMS CODE, SUPERCODE

Power Balance (1D)

- Transport governed by the Rebut-Lallia-Watkins Critical Temperature Gradient model.

Particle Balance (0D)

- $n_j(0)/\langle n_j \rangle = 1.26, \quad j = \mathrm{D,T},\alpha,\mathrm{Be}.$
- $\tau_\alpha/\tau_E = 17.$
- $\langle n_{Be} \rangle/\langle n_e \rangle = 0.01.$

Current Balance (0D)

- $I_p = I_{FW} + I_{NB} + I_{BS}, \; q_{MHD}(0) = 1.01$ (j–profiles not self-consistent).

MHD Equilibrium (2D)

- Variational solution including X-point geometry.
- Consistent with pressure profiles but not current profiles.

FWCD SCENARIOS USING ITER CDA CURRENT DRIVE FIGURE-OF-MERIT

Parameter	$W = 0.25 \mathrm{MW/m^2}$	$W = 0.5 \mathrm{MW/m^2}$	$W = 0.75 \mathrm{MW/m^2}$
Q	5.14	4.57	4.27
P_{fus} [MW]	429	861	1291
P_{FW} [MW]	83.6	189	303
I_p [MA]	8.12	15.6	22.0
$\langle T_e \rangle_n$ [keV]	13.5	19.7	23.0
$T_e(0)$ [keV]	51.2	62.8	69.3
γ_{FW} [10^{20} $\mathrm{m^{-2}}$ MA / MW]	0.183	0.257	0.297
f_{BS} [%]	50.6	39.5	32.7
$\langle n_e \rangle$ [10^{20} / $\mathrm{m^3}$]	0.492	0.664	0.815
τ_E / τ_{ITER-P}	3.91	3.01	2.51

$$\gamma_{FW} = 0.046 \frac{\langle T_e \rangle_n}{2 + Z_{eff}}$$

NB CURRENT DRIVE SCENARIOS
(E = 500 keV, R_{tan} = 6m)

Parameter	$W = 0.25 MW/m^2$	$W = 0.5 MW/m^2$	$W = 0.75 MW/m^2$
Q	6.38	4.46	3.93
P_{fus} [MW]	430	862	1295
P_{NB} [MW]	67.3.	193	330
I_{tp} [MA]	7.87	16.4	23.1
$\langle T_e \rangle_n$ [keV]	12.1	18.8	22.2
$T_e(0)$ [keV]	40.1	48.8	51.9
$\gamma_{NB}[10^{20}\ m^{-2}\ MA/MW]$	0.219	0.247	0.259
f_{BS} [%]	45.2	33.0	28.7
$\langle n_e \rangle$ $[10^{20}/m^3]$	0.441	0.562	0.671
τ_E/τ_{ITER-P}	3.68	2.63	2.14

NB CURRENT DRIVE SCENARIOS
(E = 1300 keV, R_{tan} = 6m)

Parameter	$W = 0.25 MW/m^2$	$W = 0.5 MW/m^2$	$W = 0.75 MW/m^2$
Q	8.74	7.26	6.49
P_{fus} [MW]	429	862	1295
P_{NB} [MW]	49.1	118	199
I_p [MA]	7.20	14.2	19.9
$\langle T_e \rangle_n$ [keV]	10.0	15.4	18.0
$T_e(0)$ [keV]	36.6	45.9	50.6
$\gamma_{NB}[10^{20}\,m^{-2}\,MA/MW]$	0.311	0.386	0.405
f_{BS} [%]	46.9	35.4	31.5
$\langle n_e \rangle$ $[10^{20}/m^3]$	0.512	0.646	0.767
τ_E/τ_{ITER-P}	3.75	2.77	2.26

New *SUPERCODE* Runs for TPX

- ST and AT performance runs for 3.35T baseline and 4T candidate design under:
  (a) Day 1 conditions (P_{nb}=8MW, P_{ic}=8MW)
  (b) Upgrade conditions (P_{nb}=16MW, P_{ic}=12MW)

- New 4T candidate sensitivities: Meeting the ST requirements of β_N=3.5% with day 1 powers by:
  (a) dropping B and I, (b) enhanced confinement

- Comparison of higher field, higher current machines within the same geometry envelope (R=2.25m, A=4.5): Why not go for 4.75T and 2.6MA?

- Maximum field and current for a day 1 cost ceiling of $440M in optimized (not fixed geometry): Why not go for 5.1T, 2.7MA?

- The impact of minimum bootstrap collisionality requirements on multi-modal design points: ≤10% costs you big!

- Multi-modal optimized designs: minimum cost machines that <u>do</u> meet the mission (they're constrained to).

Typical Applied Constraints

PHYSICS AND OPERATIONAL CONSTRAINTS

- β_N =3.5 (ST) / 3.0(AT), • H = 2.0 (quadrature DIII and $H\tau_{ITER-P}$), • κ_X = 2.0,
- $V^* \leq 0.3$ (monitored but not constr.) • BS collisionality correction ≤ 10%,
- $V_{fast}\,\tau_E \geq 8$, • $\langle n \rangle \leq I/\pi a^2$ (Greenwald), • $\langle n \rangle \geq 0.3\, n_{Greenwald}$,
- $\langle n \rangle \geq n_{NB} = (dN_{NB}/dt).2\tau_E/(1 - 0.5)V$, • $n(0).a \geq 0.4e20m^{-3}*0.67m$
- V•sec = full ind ramp(C_{ejima}=0.6) + 2V-sec flattop +2V-sec,
- For AT: $1.05 \leq q_0 \leq 1.30$ (first stability regime), $q_{95} \geq 3.0$, $q^* \geq 4.5$
- For ST: q_0 =1.05 and q_{95} = 3.0, • Separate electron and ion power bal.
- $\beta_{fast}/\beta \leq 0.25$, • $\varepsilon*\beta_p \leq 1.0$ (first stability regime, monitored only).
- New, density-dependent impurity specs
- $R_{tan,NB}$ = R - a/2, except where noted; NB shinethrough ≤ 5%; beam footprint consistent with geometry
- $I = I_{NB} + I_{IC} + I_{BS}$, where I_{IC} can be <0 if req'd.
- Heating/CD: Day-1: P_{NB} = 16 MW(co) ,P_{IC} = 12 MW; Full complement: P_{NB} =16(co)+8(count.)+8(co) MW, P_{IC} = 18 MW.
- $-1 \leq f_{IC} \leq 1$, where $I_{IC} = f_{IC}\, P_{IC}\, \gamma_{IC}/nR$.
- Goldston TF ripple constraint (this or beam duct constraint determines TF outer leg)
- Divertor conditions: unconstrained, except where noted.
- Profiles: For ST: α_n= 0.31, α_T= 1.29. For AT: α_n= 0.75, α_T= 0.93 (variable in 1-1/2D if χ's supplied).

GENERAL ENGINEERING CONSTRAINTS

Optimization runs have engineering design constraints for:
PF, TF, builds, ripple, port access,.... ,etc. (examples shown later.)
Eng. variables for baseline runs --stresses, builds, etc. -- are fixed.

SYSTEMS VARIABLES FOR OPTIMIZATION RUNS

R, a, (A), I, q0, q95, B_o, n_e, n_i, T_e, T_i, f_{BS}, P_{NB}, P_{IC}, f_{IC}, Δ_{TF}, B_{TF}, Δ_{OH}, builds, etc. . . .(Profiles also variable in 1-1/2D.)

COMPARISON OF THE 3.35T BASELINE AND 4T CANDIDATE: AT PERFORMANCE (Minimum BS collisionality)

	Day 1 Powers (8MW NB, 8MW IC)		Day N Powers (16MW NB, 12MW IC)	
	3.35T Baseline	4T Candidate	3.35T Baseline	4T Candidate
BS coll correction (min)	28.9%	36.4%	18.0%	15.8%
f_{BS} / $f_{BS,NC}$	0.667* / 0.938	0.667* / 1.05	0.667* / 0.813	0.667* / 0.792
I (MA)	0.490	0.486	0.639	0.726
q_{95} / q_0	11.0 / 1.3*	12.7 / 1.3*	8.91 / 1.3*	9.02 / 1.3*
B (T)	3.35*	4.0*	3.35*	4.0*
β_N(%)	3.0*	3.0*	3.0*	3.0*
T_e / T_i (keV)	3.55 / 2.97	3.27 / 3.09	5.00 / 3.85	6.13 / 5.34
n_e (10^{20} m^{-3})	0.306*	0.361	0.306*	0.306*
P_{NB} used / install. (MW)	4.27 / 8	6.71 / 8	4.05 / 16	6.05 / 16
P_{IC} used / install. (MW)	8* / 8	8* / 8	10.6 / 12	12* / 12
f_{IC}	-0.408	-0.787	-0.191	-0.301
Div. pow den. (MW/m^2)	3.05	3.26	4.41	5.56
Div. temp (eV)	364	394	385	483
Key Constraints[a]:				
$f_{BS} \geq 0.67$ – Min	0*	0*	0*	0*
$T_e/T_i \leq 30\%$ - Max	0.081	0.184	0*	0.116
q* – Min	0.409	0.495	0.254	0.287
n_0.3Green. – Min	0.388	0.486	0.203	0.094
n*a neut. pentr. – Min	0*	0.152	0*	0*
n_NB fuel. – Min	0.758	0.689	0.738	0·591
$\nu_{fast} \cdot \tau_E$ – Min	0.095	0.197	0.221	0.265
$\beta_{NB,\ fast}$ – Max	0.075	0*	0.210	0.58

(a) Constraints are shown as: residual(x) = 1 – x/xmax, or = 1 – xmin/x (residual = 0 at constr. bound, NA = constraint not active). *-- variable at a constraint bound or fixed

COMPARISON OF THE 3.35T BASELINE AND 4T CANDIDATE: ST PERFORMANCE (Maximum β_N)

	Day 1 Powers (8MW NB, 8MW IC)		Day N Powers (16MW NB, 12MW IC)	
	3.35T Baseline	4T Candidate	3.35T Baseline	4T Candidate
β_N(%) (maximum)	2.29	1.93	3.31	2.81
I (MA)	1.83	2.0*	1.87	2.0*
B (T)	3.35*	4.0*	3.35*	4.0*
f_{BS} / $f_{BS,NC}$	0.220 / 0.238	0.199 / 0.215	0.319 / 0.352	0.294 / 0.328
q_{95} / q_0	3.0* / 1.05*	3.25 / 1.05*	3.0* / 1.05*	3.30 / 1.05*
T_e / T_i (keV)	7.63 / 8.86	8.26 / 9.68	7.38 / 8.87	7.45 / 8.70
n_e (10^{20} m^{-3})	0.368(e)	0.380(c)	0.562(e)	0.634(c)
$n_i(0) : \tau_E . T_i(0)$ (10^{20} m^{-3} s keV)	1.24	1.58	1.69	2.12
P_{NB} used / install. (MW)	8* / 8	8* / 8	16* / 16	16* / 16
P_{IC} used / install. (MW)	8* / 8	8* / 8	12* / 12	12* / 12
f_{IC}	0.831	1.0*	0.402	0.777
Div. pow den. (MW/m^2)	7.40	7.02	13.3	12.1
Div. temp (eV)	211	206	227	192
Key Constraints(a):				
n 0.3Green. – Min	NA (-0.90)(e)	NA (-1.0)(c)	NA (-0.27)(e)	NA (-0.20)(c)
n*a neut. pentr. – Min	0*(b)	0*(b)	0·272	0.354
n_NB fuel. – Min	0.122	0.043	0*	0*
$\nu_{fast}.\tau_E$ – Min	0.689	0.734	0.762	0.813
$\beta_{NB,\,fast}$ – Max	0.272	0.346	0.372	0.491

(a) Constraints are shown as: residual(x) = 1 – x/xmax, or = 1 – xmin/x (residual = 0 at constr. bound, NA = constraint not active). (b) Limit lowered by 10% to get a solution . (c) $0.3^*n_{Greenwald}$ $=0.76e20m^{-3}$ cannot be met. (d) $0.3^*n_{Greenwald}$ $=0.50e20m^{-3}$ cannot be met. (e) $0.3^*n_{Greenwald}$ $=0.70e20m^{-3}$ cannot be met

* -- Variable at a constraint bound or fixed

4T CANDIDATE DESIGN: ST SCENARIO SENSITIVITIES

	Day-1 powers — Max. β_N	Day-N powers — Max. β_N	β_N=3.5 with day-1 powers. — Reduced B,I	β_N=3.5 with day-1 powers. — Enhanced confinement
β_N(%) (maximum)	1.93	2.81	3.5*	3.5*
I (MA)	2.0*	2.0*	1.30	2.0*
B (T)	4.0*	4.0*	2.31	4.0*
f_{BS} / $f_{BS,NC}$	0.199 / 0.215	0.294 / 0.328	0.318 / 0.352	0.436 / 0.464
q_{95} / q_0	3.25 / 1.05*	3.30 / 1.05*	3.0* / 1.05*	3.35 / 1.05*
Confinement H factor	2.0	2.0	2.0	3.71(b)
T_e / T_i (keV)	8.26 / 9.68	7.45 / 8.70	5.61 / 6.13	11.6 / 11.70
n_e (10^{20} m^{-3})	0.380(c)	0.634(c)	0.368(d)	0.598
P_{NB} used / install. (MW)	8* / 8	16* / 16	8* / 8	5.04 / 8
P_{IC} used / install. (MW)	8* / 8	12* / 12	8* / 8	8* / 8
f_{IC}	1.0*	0.777	+0.340	+1.0
Div. pow den. (MW/m^2)	7.02	12.1	7.53	4.25
Div. temp (eV)	206	192	216	63
Key Constraints(a):				
n 0.3Green. – Min	NA (-1.0)(c)	NA (-0.20)(c)	NA (-0.346)(d)	NA (-0.278)(c)
n*a neut. pentr. – Min	0.0*(f)	0.354	0*(b)	0.316
n NB fuel. – Min	0.043	0*	0.374	0*
$\nu_{fast}.\tau_E$ – Min	0.734	0.813	0.555	0.935
$\beta_{NB,\ fast}$ – Max	0.346	0.491	0.117	0.844

(a) Constraints are shown as: residual(x) = 1 – x/xmax, or = 1 – xmin/x (residual = 0 at constr. bound, NA = constraint not active).
(b) H*ITER-P scaling; no DIII quad because of saturation. (c) 0.3*$n_{Greenwald}$ =0.76e20m^{-3} cannot be met. (d) 0.3*$n_{Greenwald}$ =0.50e20m^{-3} cannot be met.
(e) 0.3*$n_{Greenwald}$ =0.70e20m^{-3} cannot be met (f) Limit reduced 10% to get solution * -- Variable at a constraint bound or fixed

COMPARISON OF HIGHER FIELD, HIGHER CURRENT MACHINES

	SSAT 3.35T BASELINE		4T CANDIDATE. —	HIGHER FIELD MACHINES AT R=2.25m, A=4.5, q=3.0			5T, 2MA MACHINE
	Benchmark	Min. Cost	Min. Cost	4.25T	4.5T	4.75T[a]	
Day-1 Cost (M$)[d]	381	365	385	399	411	424	423
Day-N Cost (M$)[e]	398	382	402	416	428	441	440
R (m)	2.25	←	←	←	←	←	←
a (m)	0.5	←	←	←	←	←	←
A	4.5	←	←	←	←	←	←
I (MA)	1.85	←	2.0	2.33	2.46	2.59	2.0
B (T)	3.35	←	4.0	4.25	4.50	4.75	5.0
q_{95}	3.0	←	3.28[b]	3.0	3.0	3.0	4.05
$n_i(0) \cdot \tau_E \cdot T_i(0)$ (10^{20} m^{-3} s keV)[c]	1.69	←	2.12	2.75	3.10	3.47	2.45

(a) Maximum field which can be accommodated in present geometry at q95=3.0 (b) q95=3.25 - 3.30 depending on whether day 1 or day N powers are used (c) Evaluated with day N powers (16MW NB, 12MW IC) (d) 8/8/1.5MW (e) 16/12/1.5MW

WHAT CANDIDATE DESIGN HAS THE MAXIMUM ATTAINABLE TOROIDAL FIELD FOR A DAY-1 COST OF $440M ?

- Maximize the toroidal field with all design parameters varying subject to:
  - Current I ≥ 2MA;
  - Day 1 cost ≤ $440M;
  - Powers: 8MW NB, 8MW IC, 1.5MW LH

	Optimum A, q95 ≥ 3.0	Fixed A = 4.5, q95 ≥ 3.0	Fixed A = 4.5, q95 = 3.0
Maximum B (T)	5.78	5.36	5.10
Day 1 costs (M$)	440	←	←
Day N costs (M$)	457	←	←
I(MA)	2.0*	2.0*	2.66
A	5.40	4.5*	4.5*
q_{95}	3.0*	4.18	3.0*
R(m)	2.15	2.21	2.17
a(m)	0.397	0.490	0.483

MULTI-MODAL OPTIMIZATION CAPABILITY HAS BEEN ADDED TO THE SYSTEMS CODE

This means that we can now find optimized machines that meet the mission requirements for multiple operating modes.

We've used this new capabilty to perform some new parametric studies for SSAT:

- Examination of the effect of varying the maximum allowable collisional correction to the bootstrap current in the AT mode.
- Examination of minimum cost machines that satisfy the AT and ST missions at various aspect ratios.
- Examination of minimum cost machines with reduced toroidal field in the ST mode.
- Examination of minimum cost machines with increased plasma currents in the AT mode.

MULTI-MODAL OPTIMIZATIONS ARE MUCH MORE COMPLICATED THAN UNI-MODAL OPTIMIZATIONS

The problem is still that of constrained optimization.

However, there are several new issues that must be addressed:

- New types of variables are needed.
- New types of constraints are needed.
- The scale of the problem is greatly increased.
- How do we do it?

NEW TYPES OF VARIABLES ARE NEEDED FOR MULTI-MODAL OPTIMIZATIONS

Modal variables

These are variables that are duplicated in each mode. Examples:

Density, temperature, plasma current, NB/ICRH powers, PF coil currents

Synchronized variables

These are variables that have the same value in all modes. Examples:

Major radius, TF coil build, TF/PF coil composition

Composite variables

These are variables whose values depend on all the modes. The value is that of the largest modal variable. Examples,

Installed NB/ICRH powers, cryo bldg volume, ICH/FWCD cost (WBS 23)

NEW TYPES OF CONSTRAINTS ARE NEEDED FOR MULTI-MODAL OPTIMIZATIONS

Modal Constraints

These are constraints that are duplicated in each mode. Examples:

MHD equilibrium, power balance, magnet stress, volt-seconds

Synchronized Constraints

These are constraints that apply to synchronized variables only. Examples:

Radial/vertical build, NB access

Mode-specific Constraints

These are constraints that are applied in specific modes only. Examples:

$q(0) \geq 1.05$ (ST), $q(0) \geq 1.3$ (AT), Minimum B.S. current fraction (AT)

THE MULTI-MODAL PROBLEM IS MUCH LARGER THAN THE UNI-MODAL PROBLEM

A typical SSAT optimization run consists of

- 63 variables
- 40 equality constraints
- 47 inequality constraints

Runs require about 1/2 hour of wall-clock time on the NERSC H-P server.

A TYPICAL MULTI-MODAL VARIABLE SET

#	Name	Value	Scaling	Lower Bnd	Upper Bnd	Hit	Act
108	sigma[at]	0.0728632	5	--------	--------	---	On
109	sigma1[at]	0.0929602	5	--------	--------	---	On
110	eta[at]	0.0910069	1	--------	--------	---	On
111	kappaAxis[at]	1.32998	0.5	--------	--------	---	On
112	nu[at]	1.4234	1	--------	--------	---	On
113	lZero[at]	1.47603	1	--------	--------	---	On
114	jZero[at]	-3.08586	1	--------	--------	---	On
115	plascur[at]	1	1	1	--------	L	On
116	alpha_f(1)[at]	1.0095	1	--------	--------	---	On
117	deutVarParms(1)*	0.818882	1	0.2	--------	---	On
118	deutVarParms(2)*	-0.974296	1	--------	--------	---	On
120	deutVarParms(5)*	0.839956	1	0	--------	---	On
121	carbVarParms(1)*	0.0140899	100	0	--------	---	On
122	oxyVarParms(1)[*	0.0009243	1000	0	--------	---	On
123	ironVarParms(1)*	0.000518599	10000	0	--------	---	On
124	elecTempVarParm*	11.475	0.1	0	20	---	On
125	elecTempVarParm*	-0.864524	1	--------	--------	---	On
126	ionTempVarParms*	9.07558	0.1	0	20	---	On
129	pICRHTotal[at]	17.9233	1	0.05	24	---	On
130	pNBInjectedTota*	7	1	7	24	L	On
131	c_cd_ICRH[at]	-0.095535	1	-1	1	---	On
132	joheof[at]	3.2076e+07	1e-07	1e+06	1e+08	---	On
133	johbop[at]	1.77913e+07	1e-07	1e+06	1e+08	---	On
134	sigma[st]	0.0578657	5	--------	--------	---	On
135	sigma1[st]	0.0400533	5	--------	--------	---	On
136	eta[st]	0.40051	1	--------	--------	---	On
137	kappaAxis[st]	1.45221	0.5	--------	--------	---	On
138	nu[st]	1.21572	1	--------	--------	---	On
139	lZero[st]	1.12259	1	--------	--------	---	On
140	jZero[st]	-1.86894	1	--------	--------	---	On
141	plascur[st]	2.08241	1	0.1	--------	---	On
142	alpha_f(1)[st]	-0.00476994	1	--------	--------	---	On
143	deutVarParms(1)*	1.18856	1	0.2	--------	---	On
144	deutVarParms(2)*	-0.327863	1	--------	--------	---	On
146	deutVarParms(5)*	1.19738	1	0	--------	---	On
147	carbVarParms(1)*	0.0157779	100	0	--------	---	On
148	oxyVarParms(1)[*	0.00130879	1000	0	--------	---	On
149	ironVarParms(1)*	0.00058074	10000	0	--------	---	On
150	elecTempVarParm*	16.6313	0.1	0	20	---	On
151	elecTempVarParm*	-1.1794	1	--------	--------	---	On
152	ionTempVarParms*	17.4999	0.1	0	20	---	On
155	pICRHTotal[st]	21.6539	1	0.05	24	---	On
156	pNBInjectedTota*	22.3035	1	0.08	24	---	On
157	c_cd_ICRH[st]	0.916003	1	-1	1	---	On
158	joheof[st]	3.95027e+07	1e-07	1e+06	1e+08	---	On
159	johbop[st]	3.4564e+07	1e-07	1e+06	1e+08	---	On
160	rmajor[sync]	2.15525	1	0.1	10	---	On
161	rminor[sync]	0.448451	1	0.1	10	---	On
162	bt[sync]	4.49911	1	0.5	15	---	On
163	thwcndut[sync]	0.00125158	1000	0.0001	0.01	---	On
164	oacdtf[sync]	2.75236e+07	1e-07	1e+06	1e+08	---	On
165	thkcas[sync]	0.092796	3	0.03	0.12	---	On
166	gap_stfo[sync]	0.302079	1	0.1	10	---	On
167	cpttf[sync]	24365.5	0.0001	1000	60000	---	On
168	fcutf[sync]	0.746578	1	0.1	0.9	---	On
169	tdump_tf[sync]	4.80805	0.1	1	--------	---	On
170	tfcthi[sync]	0.269163	1	0.15	--------	---	On
171	dstrand[sync]	0.000480204	1000	0.0001	0.01	---	On
172	fstrandcu[sync]	0.0573228	1	0.01	0.99	---	On
173	ohcbore[sync]	0.767789	1	0.1	--------	---	On
174	ohcth[sync]	0.0976925	1	0.05	--------	---	On
175	scrapli[sync]	0.0955552	10	0.0736	1	---	On
176	rDuct_guess[syn*	3.86233	1	0.1	6	---	On

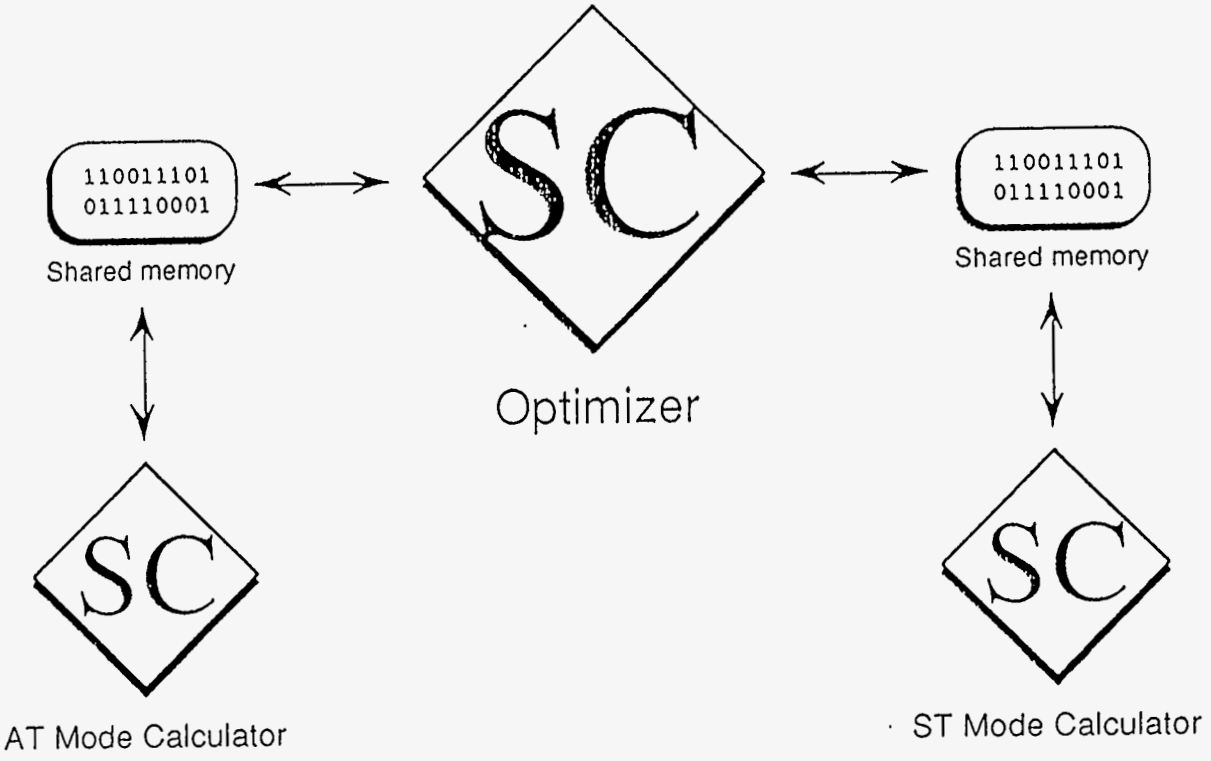
A TYPICAL MULTI-MODAL CONSTRAINT SET

i	Name	Value	Scaling	Type	Hit	Con	Act
96	l_sigma[at]	1.14211e-07	1	Equality	-----		On
97	l_sigmal[at]	5.527e-08	1	Equality	-----		On
98	l_eta[at]	-8.1621e-09	1	Equality	-----		On
99	l_kappaAxis[at]	1.33751e-08	1	Equality	-----		On
100	l_nu[at]	-1.70165e-09	1	Equality	-----		On
101	l_current[at]	-5.0902e-09	1	Equality	-----		On
102	l_energy[at]	-2.10295e-11	1	Equality	-----		On
104	epsBetapDiff[at]	0.266343	1	InEquality	no		On
105	q0LDiff[at]	-7.4686e-09	1	Equality	-----		On
106	q95Diff[at]	0.51602	1	InEquality	no		On
107	qStarDiff[at]	-1.76036e-08	1	InEquality	yes		On
108	troyDiff[at]	5.32837e-07	1	Equality	-----		On
109	deutAlpha_nDiff*	1.56259e-11	1	Equality	-----		On
110	carbConcTPXDiff*	-1.24085e-11	1	Equality	-----		On
111	coxyDiff[at]	1.90958e-13	1	Equality	-----		On
112	ironConcTPXDiff*	-1.23868e-11	1	Equality	-----		On
113	elecVolAvgDiff[*	0.432002	1	InEquality	no		On
114	rElecAvgEdgeDif*	1.62969e-11	1	Equality	-----		On
116	elecTempBalDiff*	6.33548e-07	1	Equality	-----		On
117	ionTempBalDiff[*	8.25302e-07	1	Equality	-----		On
118	tempDiffDiff[at]	0.303007	1	InEquality	no		On
119	alpha_teDiff[at]	-4.03106e-11	1	Equality	-----		On
120	bsFractionDiff[*	-8.30976e-07	1	InEquality	yes		On
121	ssedriveDiff[at]	6.09453e-07	1	Equality	-----		On
122	fastBetaDiff[at]	0.483991	1	InEquality	no		On
123	nuFastDiff[at]	0.611411	1	InEquality	no		On
125	densGreenwaldDi*	0.666301	1	InEquality	no		On
126	densNBDiff[at]	0.607641	1	InEquality	no		On
127	minGreenwaldDif*	0.100985	1	InEquality	no		On
128	neutralDensDiff*	0.353443	1	InEquality	no		On
129	stTotalDiff[at]	1	1	InEquality	no		On
130	bsFractColDiff[*	0.565797	1	InEquality	no		On
132	rippleDiff[at]	0.171211	1	InEquality	no		On
141	ftrans_ohcDiff[*	0.214608	1	InEquality	no		On
142	ftrans_ohc_0Dif*	0.597849	1	InEquality	no		On
143	vsecDiff[at]	0.00823653	1	InEquality	no		On
144	johcofDiff[at]	0.641484	1	InEquality	no		On
145	johbopDiff[at]	0.819995	1	InEquality	no		On
151	l_sigma[st]	1.67861e-07	1	Equality	-----		On
152	l_sigmal[st]	8.12514e-08	1	Equality	-----		On
153	l_eta[st]	-1.19279e-08	1	Equality	-----		On
154	l_kappaAxis[st]	2.00657e-08	1	Equality	-----		On
155	l_nu[st]	1.51695e-09	1	Equality	-----		On
156	l_current[st]	2.71048e-09	1	Equality	-----		On
157	l_energy[st]	1.64497e-09	1	Equality	-----		On
159	epsBetapDiff[st]	0.588953	1	InEquality	no		On
160	q0LDiff[st]	1.6186e-08	1	Equality	-----		On
161	q95Diff[st]	-1.66271e-08	1	Equality	-----		On
163	troyDiff[st]	8.23104e-07	1	Equality	-----		On
164	deutAlpha_nDiff*	1.24394e-11	1	Equality	-----		On
165	carbConcTPXDiff*	-5.18341e-12	1	Equality	-----		On
166	coxyDiff[st]	-2.93765e-13	1	Equality	-----		On
167	ironConcTPXDiff*	-5.33285e-12	1	Equality	-----		On
168	elecVolAvgDiff[*	0.699724	1	InEquality	no		On
169	rElecAvgEdgeDif*	1.62969e-11	1	Equality	-----		On
171	elecTempBalDiff*	4.99117e-07	1	Equality	-----		On
172	ionTempBalDiff[*	1.48056e-06	1	Equality	-----		On
173	tempDiffDiff[st]	0.825906	1	InEquality	no		On
174	alpha_teDiff[st]	-4.39264e-11	1	Equality	-----		On
175	ssedriveDiff[st]	8.04787e-07	1	Equality	-----		On
176	fastBetaDiff[st]	0.63984	1	InEquality	no		On
177	nuFastDiff[st]	0.856796	1	InEquality	no		On
179	densGreenwaldDi*	0.696881	1	InEquality	no		On
180	densNBDiff[st]	0.0427476	1	InEquality	no		On
181	minGreenwaldDif*	0.0102908	1	InEquality	no		On
182	neutralDensDiff*	0.543387	1	InEquality	no		On
184	stTotalDiff[st]	1	1	InEquality	no		On
185	radbldDiff[st]	4.04121e-14	1	Equality	-----		On
186	rippleDiff[st]	0.196121	1	InEquality	no		On
187	btorDiff[st]	-4.50647e-07	1	Equality	-----		On
188	tfcaseDiff[st]	8.24677e-07	1	InEquality	yes		On
189	tfconDiff[st]	2.26959e-06	1	InEquality	yes		On
190	vtfDiff[st]	-1.10857e-05	1	InEquality	yes		On
191	jwptfDiff[st]	6.76e-06	1	InEquality	yes		On
192	tmargtfDiff[st]	0.32505	1	InEquality	no		On
193	oitfDiff[st]	-3.7022e-05	1	InEquality	yes		On
194	ftransitionDiff*	-7.89773e-06	1	InEquality	yes		On
195	ftrans_ohcDiff[*	1.72982e-08	1	InEquality	yes		On
196	ftrans_ohc_0Dif*	0.0916297	1	InEquality	no		On
197	vsecDiff[st]	-6.01912e-07	1	InEquality	yes		On
198	johcofDiff[st]	0.521096	1	InEquality	no		On
199	johbopDiff[st]	0.547301	1	InEquality	no		On
200	portsizeDiff[st]	-1.31505e-09	1	InEquality	yes		On
201	rDuctDiff[st]	1.27676e-14	1	Equality	-----		On
202	dStrikeVVDiff[s*	1.86295e-13	1	InEquality	yes		On

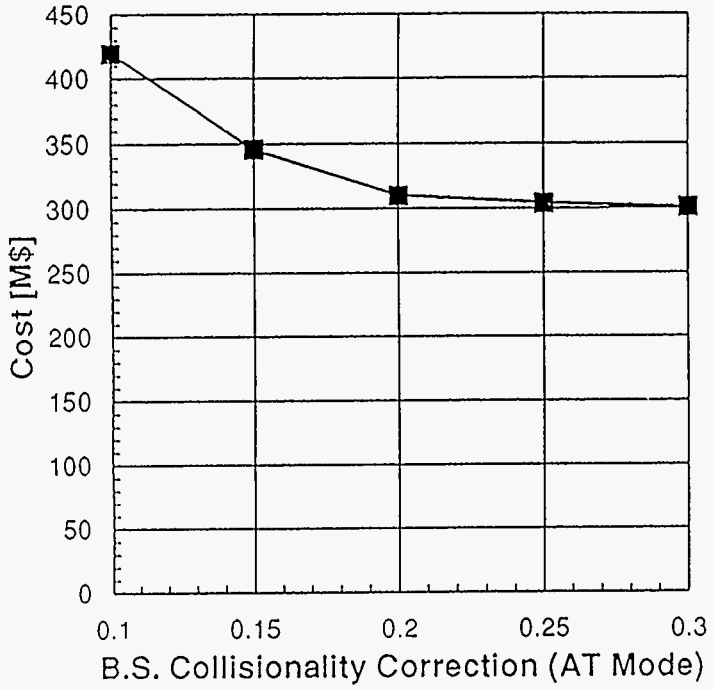
HOW DO WE DO IT?

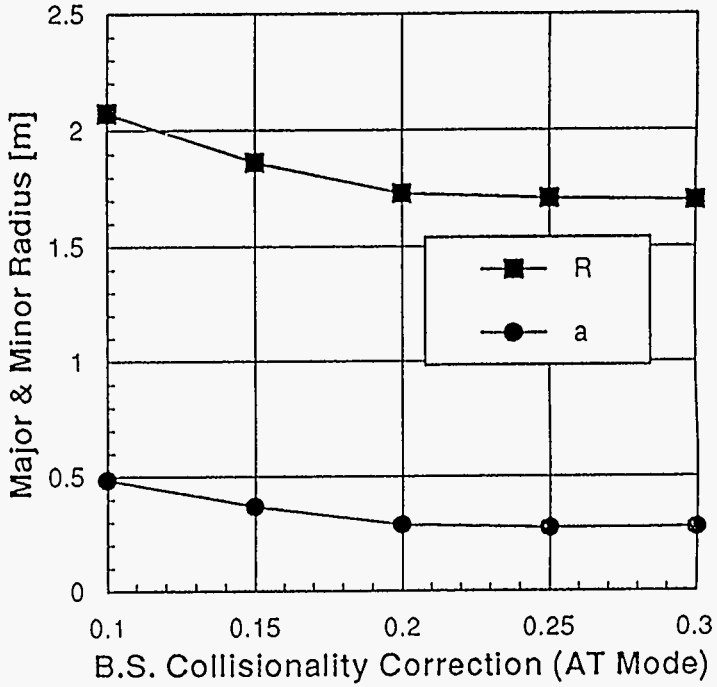
We could have directly modified all 40+ systems code modules to handle multiple modes (e.g., by changing scalars to vectors, etc).

Instead, we used the multi-tasking capabilities of UNIX to allow the systems code to clone copies of itself to compute mode quantities.



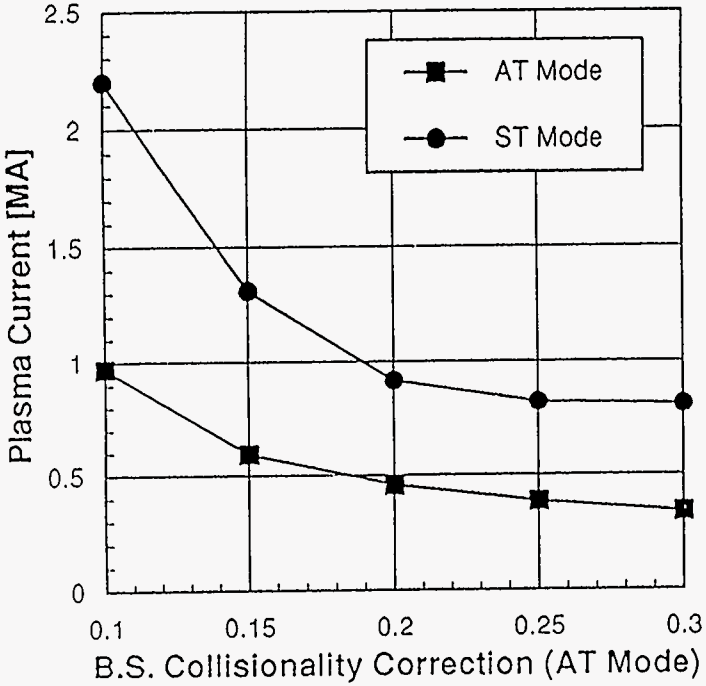
HOW MUCH DOES REDUCING THE B.S. CURRENT COLLISIONAL CORRECTION COST?

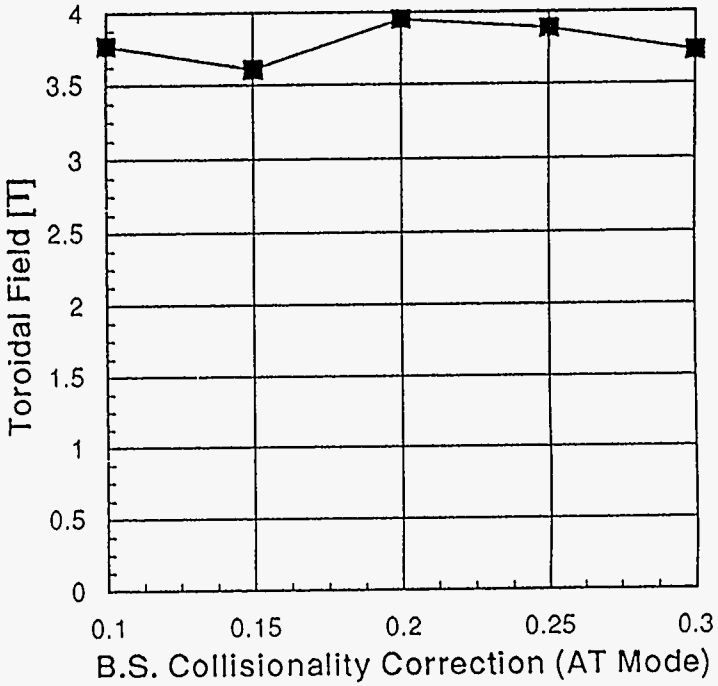




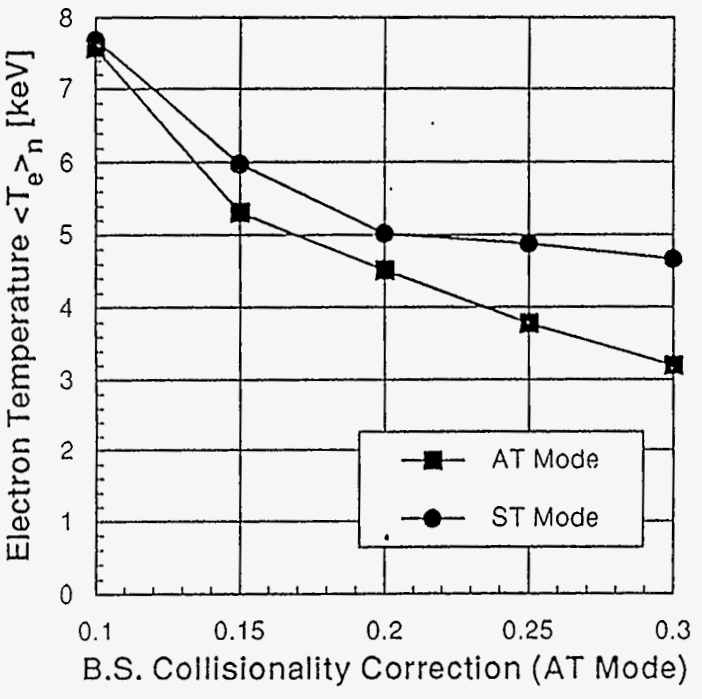
In our parametric studies, we will require designs to achieve 15% collisional correction to the bootstrap current.

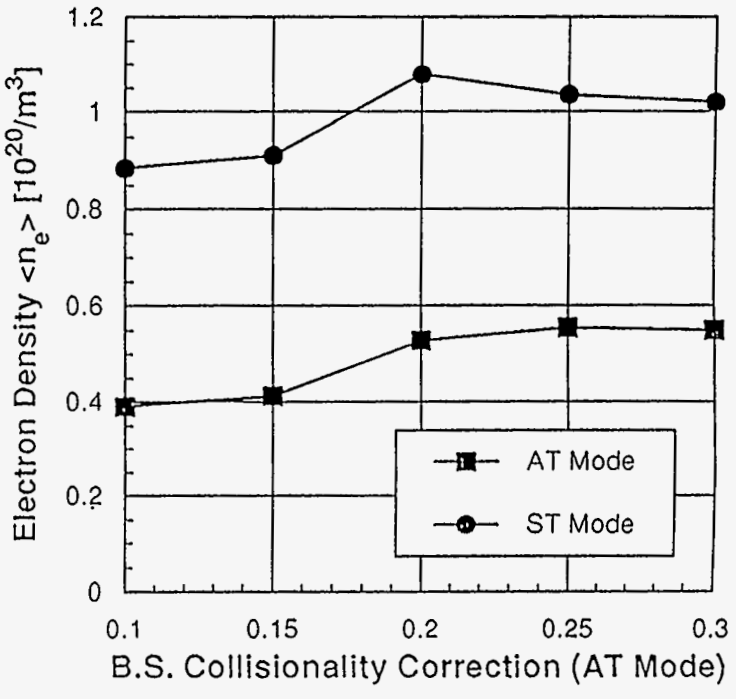
B.S. CURRENT COLLISIONAL CORRECTION PARAMETRIC SCAN: CURRENT & FIELD



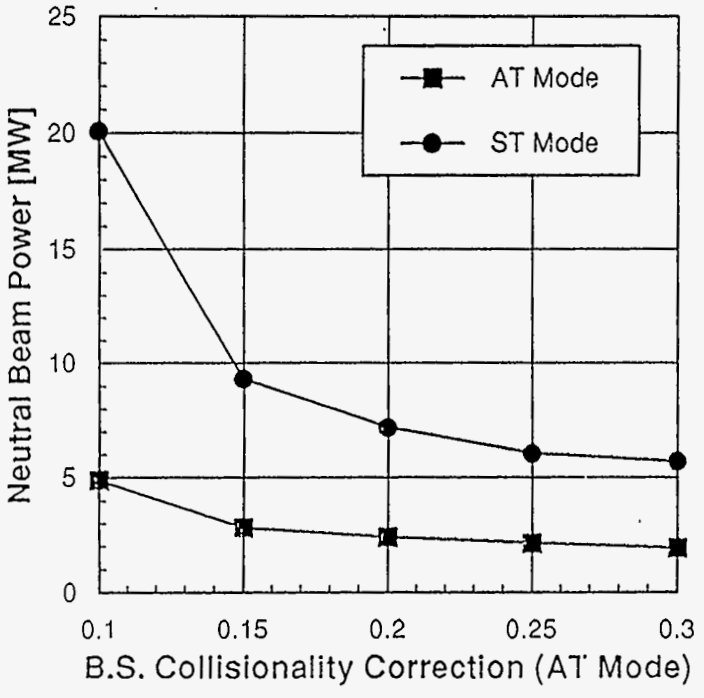


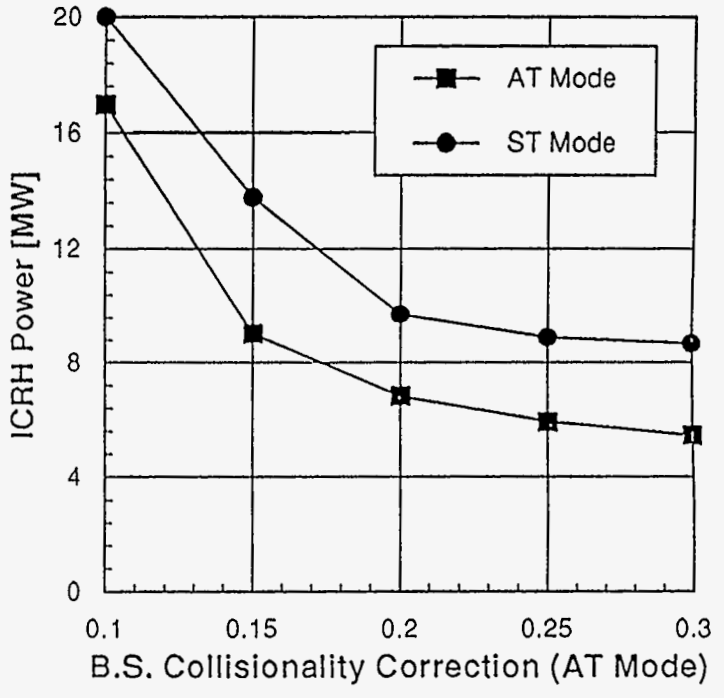
B.S. CURRENT COLLISIONAL CORRECTION PARAMETRIC SCAN: DENSITY & TEMPERATURE





B.S. CURRENT COLLISIONAL CORRECTION PARAMETRIC SCAN: HEATING

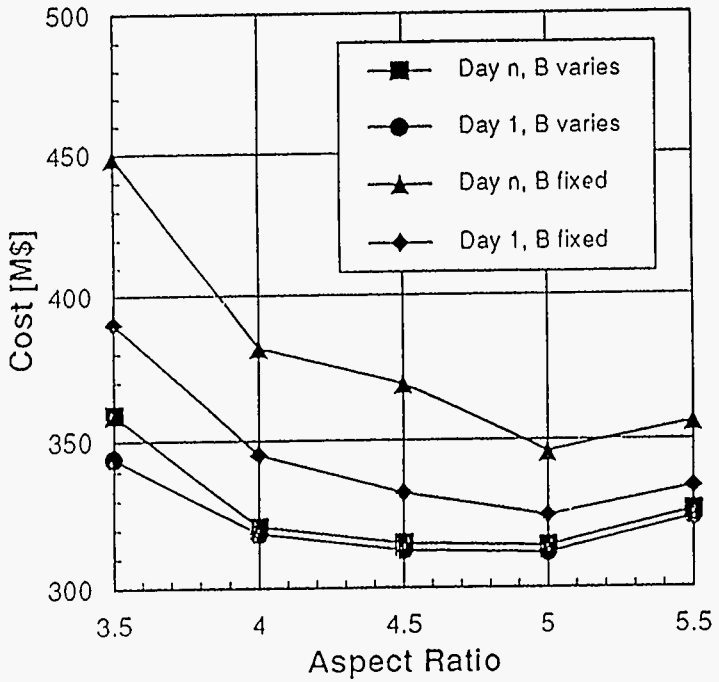


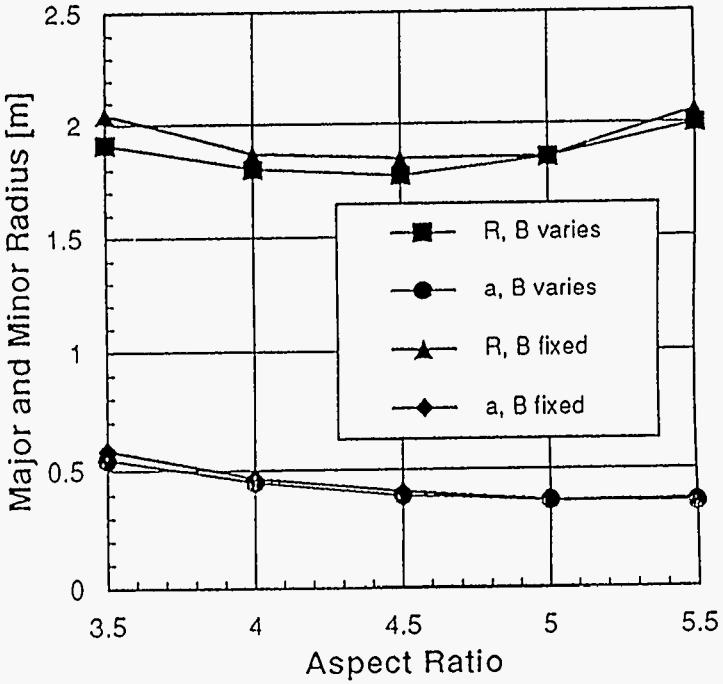


WHAT IS THE CHEAPEST MACHINE THAT SATISFIES BOTH THE AT AND ST MISSIONS?

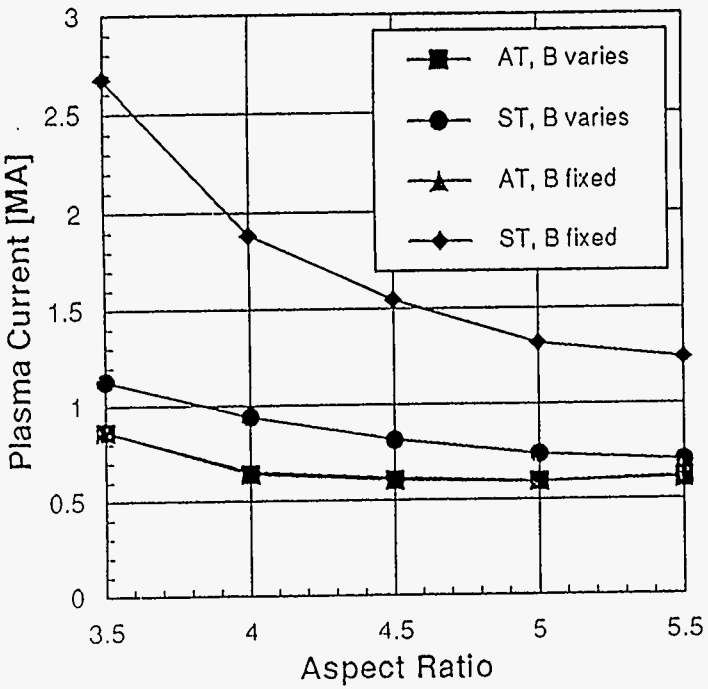
We consider two cases:

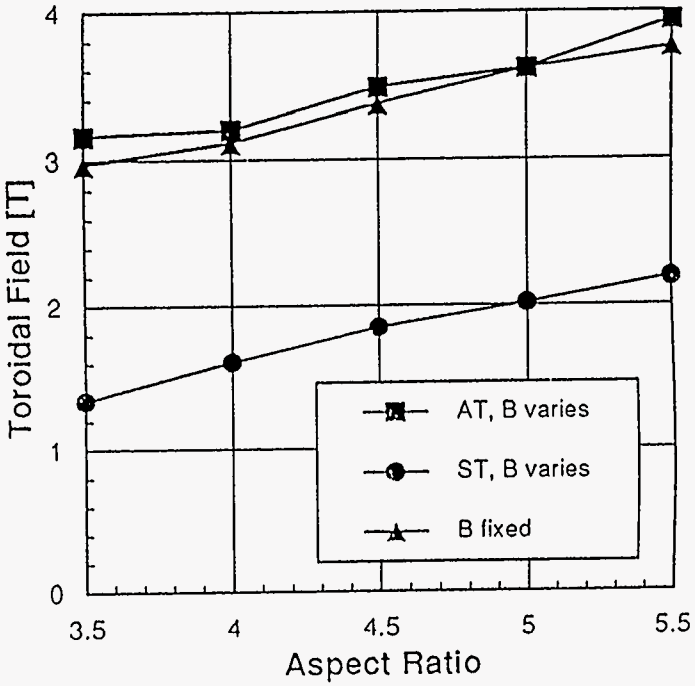
- Toroidal field same in both modes
- Field allowed to be different



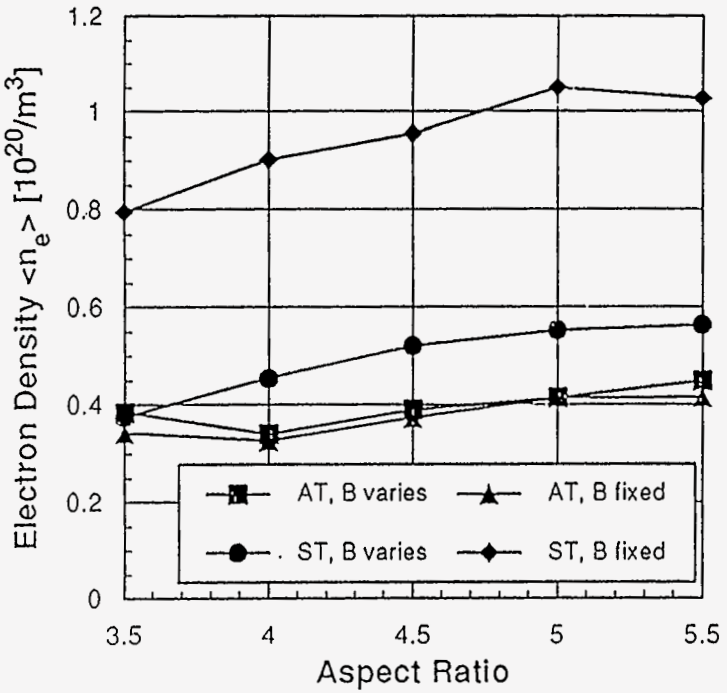


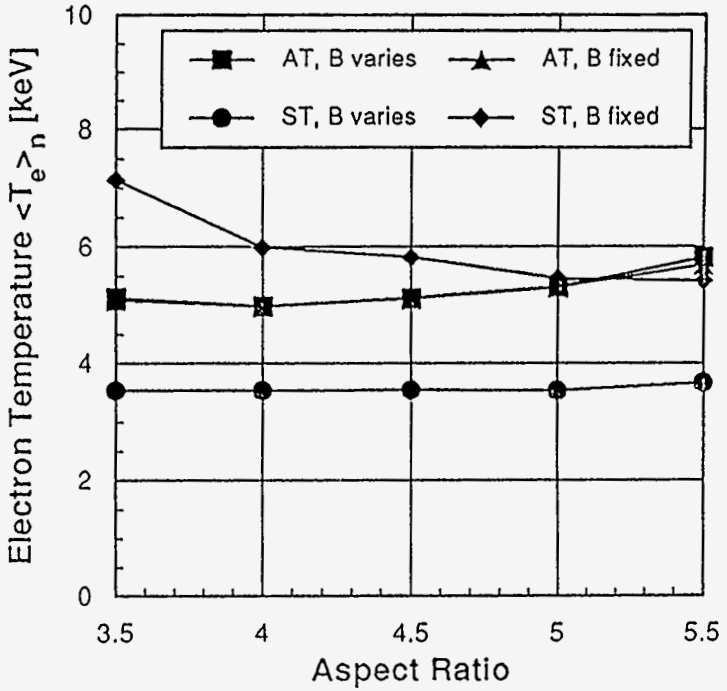
MINIMUM-COST PARAMETRIC: CURRENT & FIELD



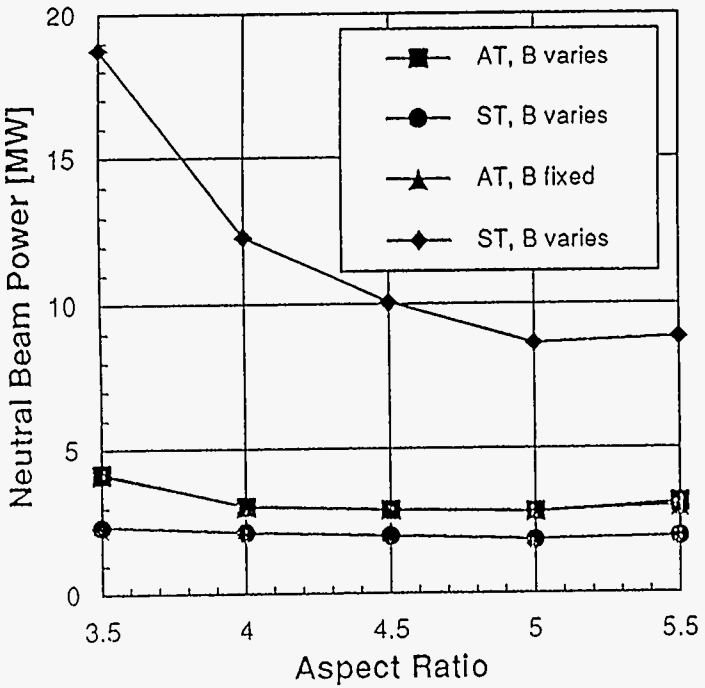


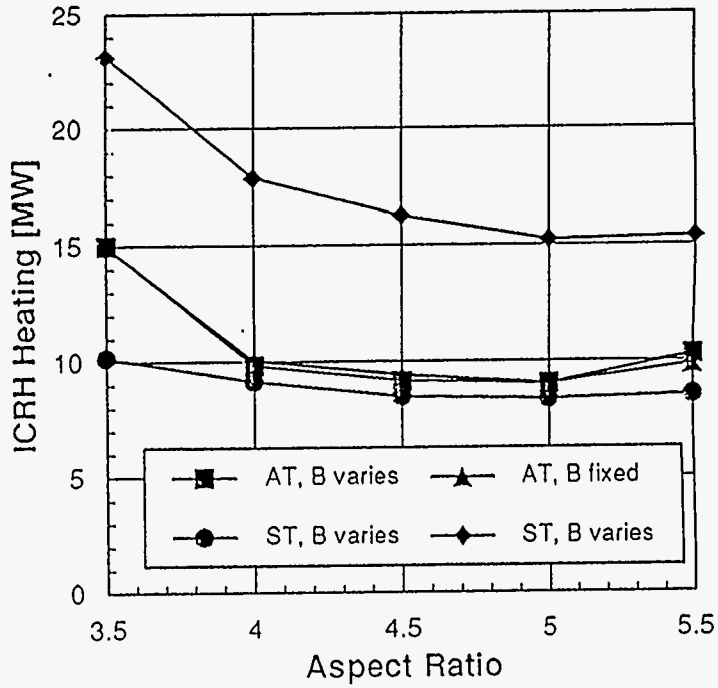
MINIMUM-COST PARAMETRIC: DENSITY & TEMPERATURE



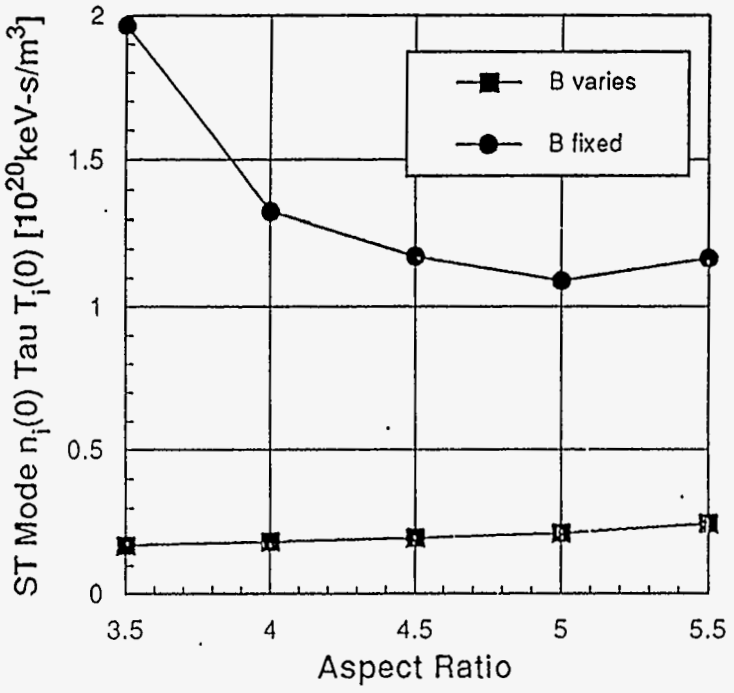


MINIMUM-COST PARAMETRIC: HEATING

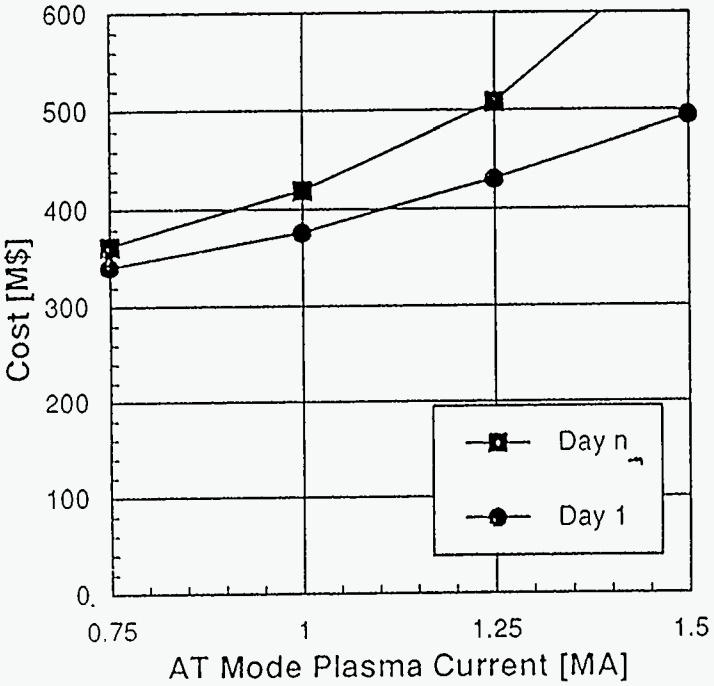


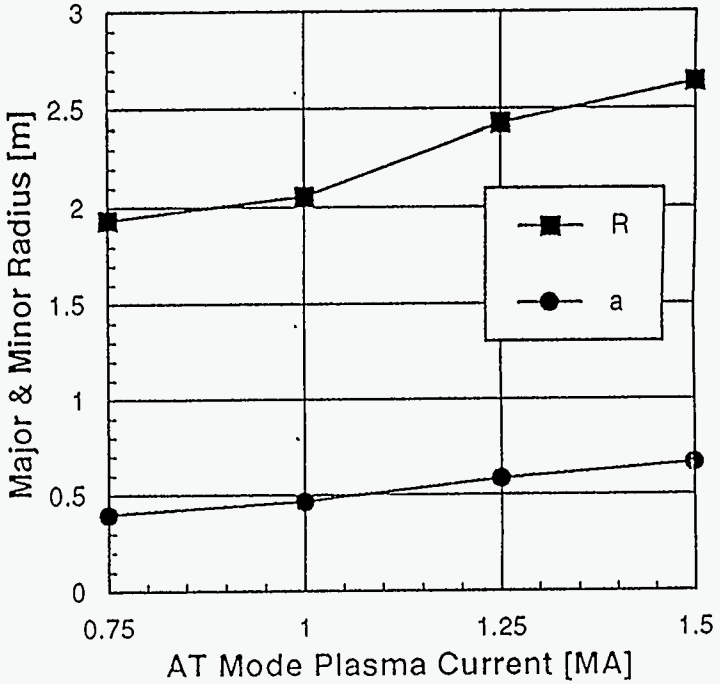


MINIMUM-COST PARAMETRIC: N-TAU-T

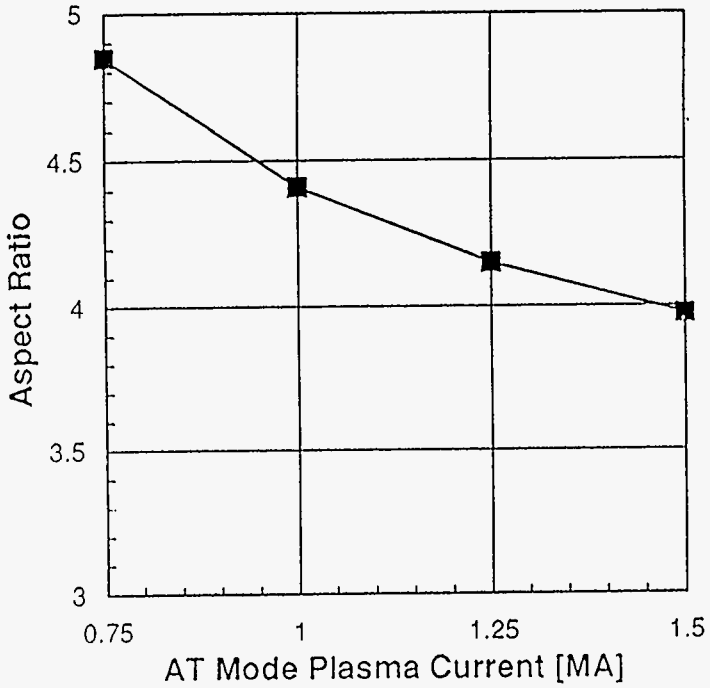


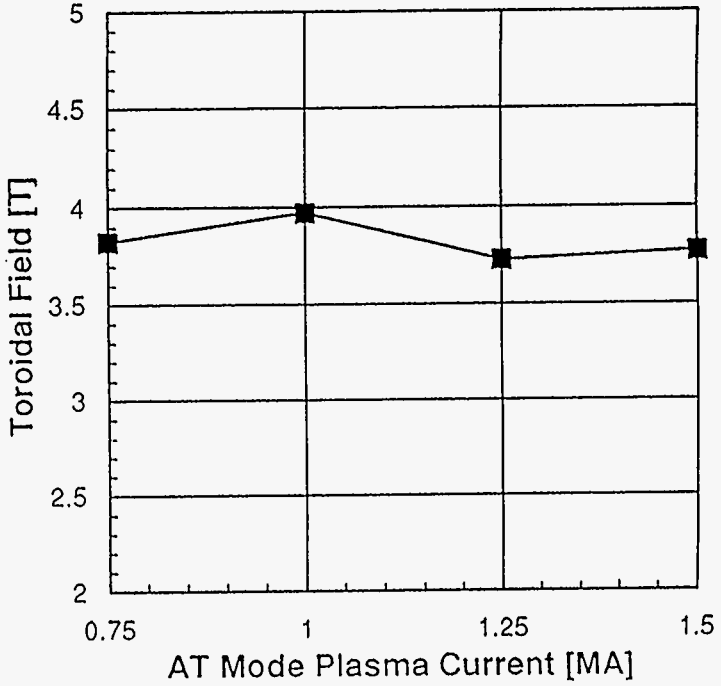
HOW MUCH DOES IT COST TO UPGRADE AT MODE PERFORMANCE?



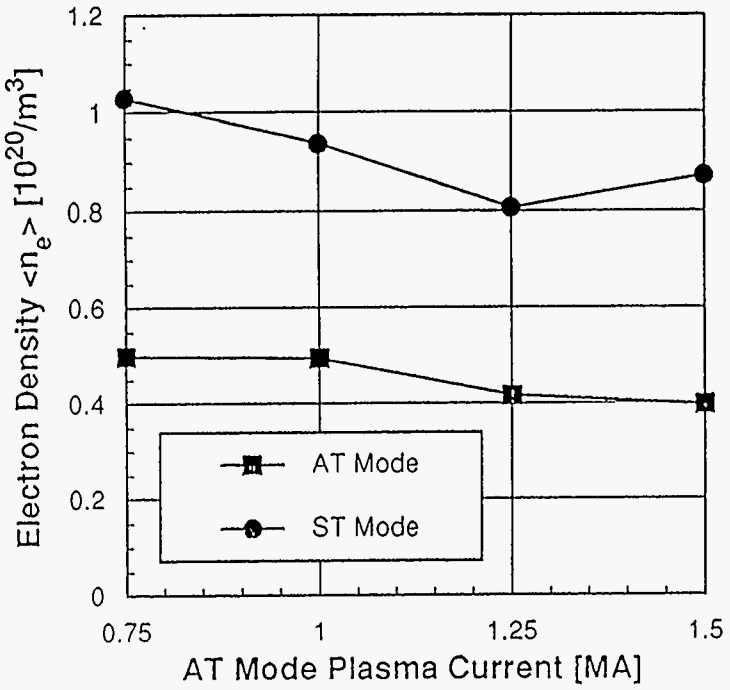


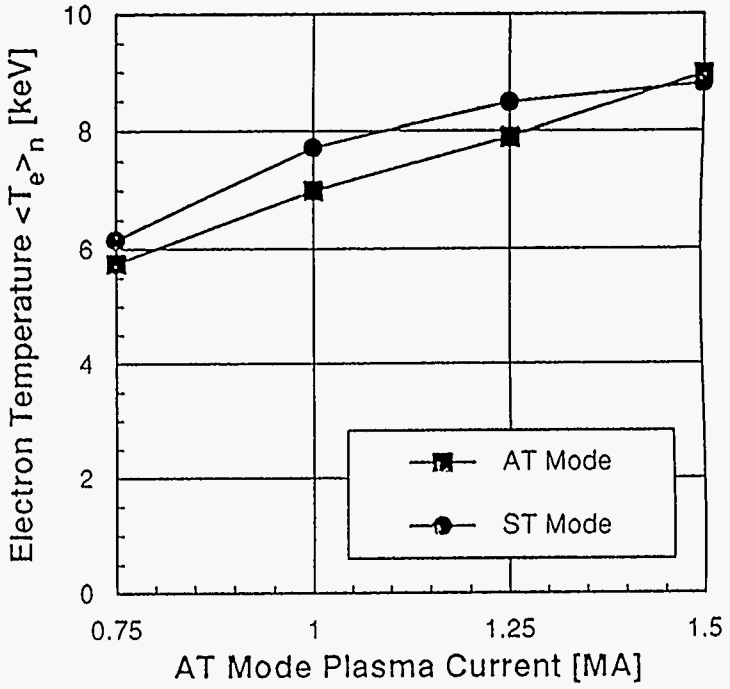
AT MODE PERFORMANCE PARAMETRIC: ASPECT RATIO & TOROIDAL FIELD



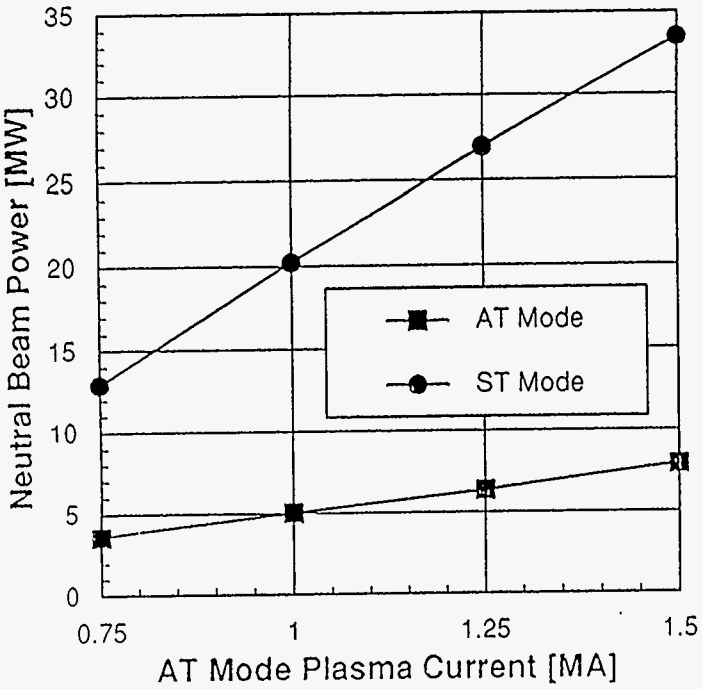


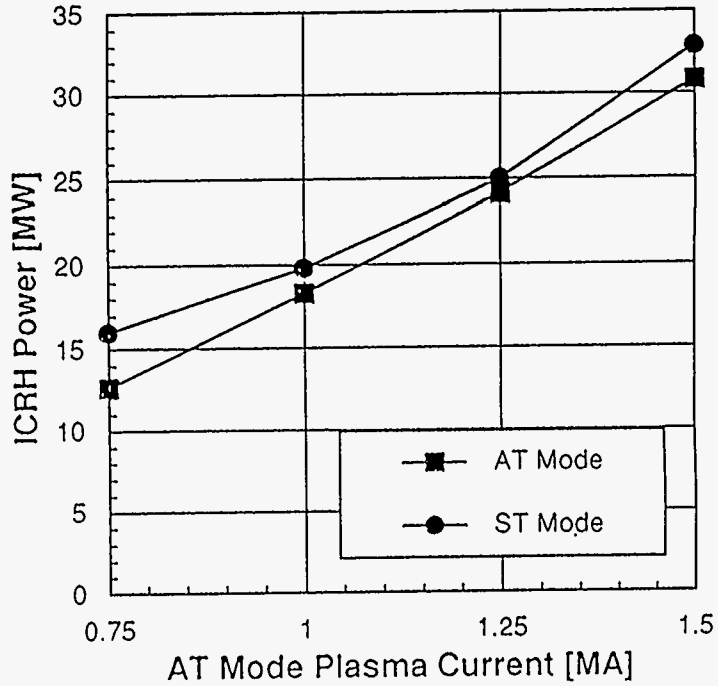
AT MODE PERFORMANCE PARAMETRIC: DENSITY & TEMPERATURE



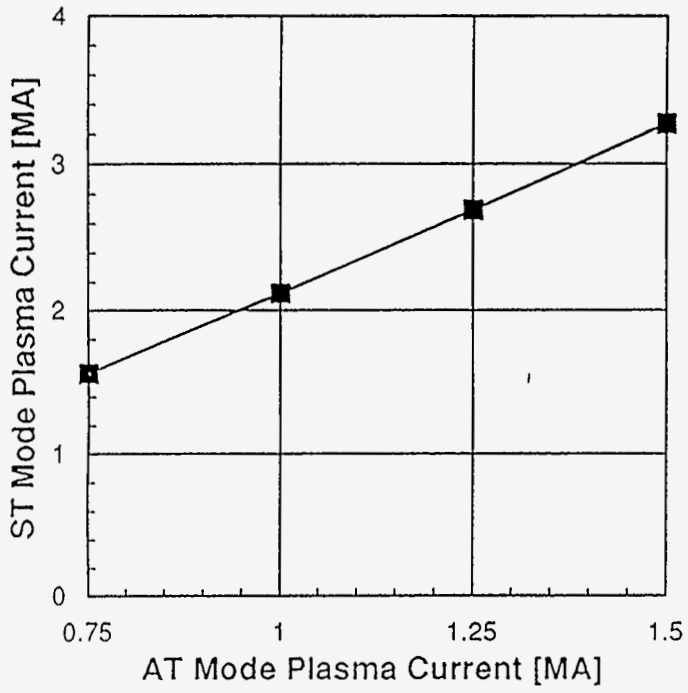


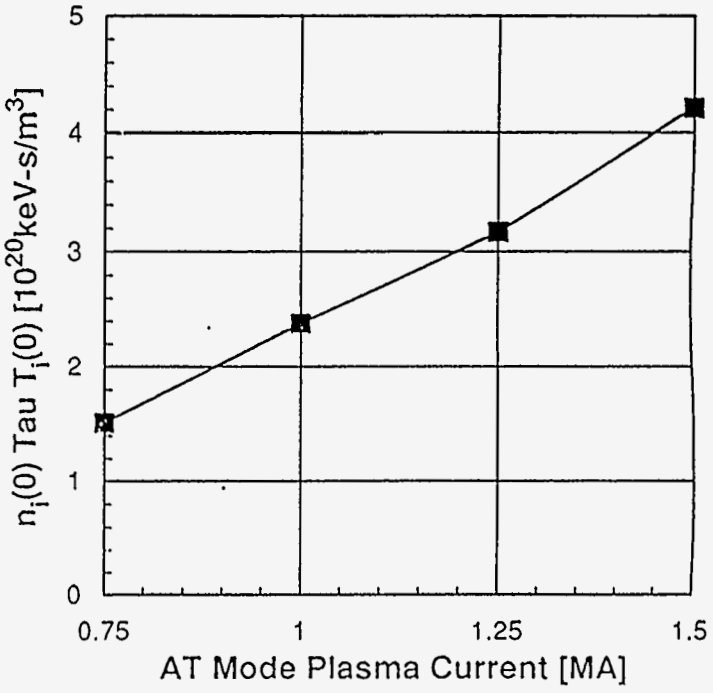
AT MODE PERFORMANCE PARAMETRIC: HEATING





AT MODE PERFORMANCE PARAMETRIC: ST MODE CURRENT & N-TAU-T





AN ATTRACTIVE MACHINE WITH GOOD AT MODE PERFORMANCE

Machine Parameters	Value	
R [m]	2.16	
a [m]	0.45	
A	4.8	
B [T]	4.5	
P_{NB} (Day n Installed) [MW]	24	
P_{IC} (Day n Installed) [MW]	24	
Day 1 Cost [M$]	397	
Day n Cost [M$]	456	
Day n Performance	AT Mode	ST Mode
I_p [MA]	1	2.08
β_p / β_t	0.015 / 3.53	0.036 / 1.98
Troyon factor	0.03	0.035
$\langle n_e \rangle / \langle n_i \rangle$ [10^{20} / m^3]	0.528 / 0.477	1.0 / 0.92
$\langle T_e \rangle_n / \langle T_i \rangle_n$ [keV]	7.19 / 5.69	8.39/ 8.84
$n_i(0)\tau_E T_i(0)$ [10^{20} kev – s / m^3]	0.71	2.87
B.S. current fraction / coll. corr.	0.67 / 0.14	0.38 / 0.12
$\nu^* / \nu_{fast}\tau_E$	0.062 / 20.6	0.127 / 55.9
P_{NB} / P_{IC} (used) [MW]	7 / 17.9	22.3 /21.7

AT MODE DAY 1 PERFORMANCE OPTIONS

With full installed heating (8 MW N.B./ 8 MW I.C.), full field/current, and H = 2 can obtain:

- B.S. current fraction of 43%, collisionality correction of 21%
- Troyon factor of 0.024
- Average electron density of 0.57×10^{20} m^{-3}
- Average density-weighted electron temperature of 4.6 keV

If H = 3.25 could be achieved at a Troyon factor of 0.035, can get:

- B.S. current fraction of 67%, collisionality correction of 15%
- Average electron density of 0.57×10^{20} m^{-3}
- Average density-weighted electron temperature of 6.7 keV

ST MODE DAY 1 PERFORMANCE OPTIONS

With full installed heating (8 MW N.B./ 8 MW I.C.) cannot drive full plasma current.

If the field is dropped to 2.35 T and the current is dropped to 1.16 MA, at H = 2 can get:

- Troyon factor of 0.035
- q(95%) = 3

If H = 2.35, B = 2.6 T, I = 1.3 MA can get:

- Troyon factor of 0.035
- q(95%) = 3
- Average electron density of 0.6×10^{20} m^{-3}
- Average density-weighted electron temperature of 4.8 keV

No more current can be driven without upgrading N.B. & I.C. systems. With 16 MW N.B, 12 MW I.C., and H = 2.8, can drive almost all the current at B = 4.15 T.

TRANSPORT MODELS IN THE SUPERCODE

- In order to use the 1½-D capabilities of the SUPERCODE, one has to specify expressions for the electron and ion energy transport coefficients. We use both semi-empirical and theory-based models.

- The semi-empirical expressions for the local χ's reproduce the global predictions of the *L*-mode confinement database,

$$\chi_j(\rho) = C_j \frac{a^2}{\tau_E} F(\rho)$$

where C_j is an adjustable coefficient, and $F(r)$ is a profile factor from experimentally-determined χ's.

- Theory-based models include a diagonal neoclassical model and Rebut's "Critical Electron Temperature Gradient Model" as follows:

$$\chi_e = \chi^{RLW}\left(1 - \frac{\left|(\nabla T_e)_c\right|}{\left|\nabla T_e\right|}\right) H\left(\left|\nabla kT_e\right| - \left|(\nabla kT_e)_c\right|\right) H(\nabla q) + \chi_{nc,e}$$

$$\chi_i = 2\chi^{RLW}\sqrt{\frac{T_e}{T_e + T_i}}\frac{Z_i}{\sqrt{1+Z_{eff}}}\left(1 - \frac{\left|(\nabla T_e)_c\right|}{\left|\nabla T_e\right|}\right) H\left(\left|\nabla kT_e\right| - \left|(\nabla kT_e)_c\right|\right) H(\nabla q) + \chi_{nc,i}$$

where

$$\chi^{RLW} = 0.5c^2(\mu_0 m_i)^{1/2}\frac{1}{B_t\sqrt{R}}\left(1-\sqrt{\varepsilon}\right)\left(1+Z_{eff}\right)^{1/2}\left|\frac{\nabla T_e}{T_e} + 2\frac{\nabla n_e}{n_e}\right|\frac{q^2}{\nabla q}\sqrt{\frac{T_e}{T_i}}$$

and

$$(\nabla kT_e)_c \equiv 0.06\sqrt{\frac{e^2}{\mu_0 m_e^{1/2}}}\frac{1}{q}\sqrt{\frac{\eta j B_t^3}{n_e (kT_e)^{1/2}}}$$

CONDITIONS FOR *SuperCode* RUNS AT IGNITION

0-D CASES WITH CDA "RULES"

- Fix ITER-R configuration (R, a, B, I, ...)
- Require 1MW/m^2 neutron wall load (P_{fusion}= 1730MW).
- Fix density and temperature profiles (α_n=0.5, α_T=1.0)
- Fix q_0 and J-profile (q_{95} determined by configuration)
- Fix alpha ash fraction at 10%
- Use CDA impurity rules: f_C, f_O, f_{Fe} = funct(n_e)
- Single fluid: $\langle T_e \rangle_n = \langle T_i \rangle_n$ = 10keV (~min. of ignition curve)
- Solve 0-D power balance at ignition (P_{aux}=0), under equilibrium conditions
- Solve for: $\langle n_e \rangle$, τ_E, $H^{(a)}$, β, β_N, z_{eff},.... etc

1-D CASES WITH REBUT ∇T_{crit} χ's

- Fix ITER-R configuration (R, a, B, I, ...)
- Require 1MW/m^2 neut. wall load (P_{fus}=1730MW) at ignition.
- Solve for q-profile (and q_0, q_{95}) consistent with ohmic profile and BS current.
- Use Rebut impurity rules: f_{Be} =1%, f_C, f_O, f_{Fe} = 0
- Determine alpha ash fractions by fixing ratio of $\tau\alpha/\tau_E$
- Solve 1-D power balance and 2-D MHD equilibrium. Auxillary power is IC with self-consistent radial deposition. Equilibrium conditions, fully evolved profiles ($\partial/\partial t = 0$).
- Solve for: T-profile, q-profile, $\langle T \rangle$, $\langle n_e \rangle$, τ_E, H, β, β_N, z_{eff},...
- From solution, can also determine the equivalent 0-D enhancement factor required after-the-fact: $H = \tau_E / \tau_{E\text{-scaling}}$

****Essence of the difference between 0-D and 1-D R-L-W is self-consistent profiles and no free parameters in the transport****

$H = \tau_E / \tau_{E\text{-scaling}}$

FORMULATION OF PILOT PLANT OPTIMIZATION SYSTEM STUDIES WITH *SUPERCODE*

- Decide on the objective function:
  e.g.: Minimize cost for a given performance requirement (may be a multi-attribute objective function)

- Decide on the performance requirement:
  e.g.: Neutron wall loading ≥ 1 *MW/m*^{2}*, and/or* $Q = \frac{P_{fusion}}{P_{aux}} \geq 1$*, etc.*

- Decide on the physics and engineering "rules":
  e.g.: Confinement, beta, q, shielding, TF/OH constraints, etc.
  (These are typically inequalities.)

- Decide on the design parameters we are free to vary; bound them if required,
  e.g.: R*,* a*,* A*,* I*,* B*,* n*,* T*,* P_{aux}*, conf'm'nt H* (≤ 2)*, beta* g_T (≤ 3.0)*, radial/vertical builds, TF/OH coil geometry, etc.*

  $\Rightarrow$ This is the formulation of a constrained, non-linear optimization problem.

  $\Rightarrow$ In *SUPERCODE*, all the performance requirements and physics/engineering rules form a global set of inequality constraints, and are solved silmultaneously.

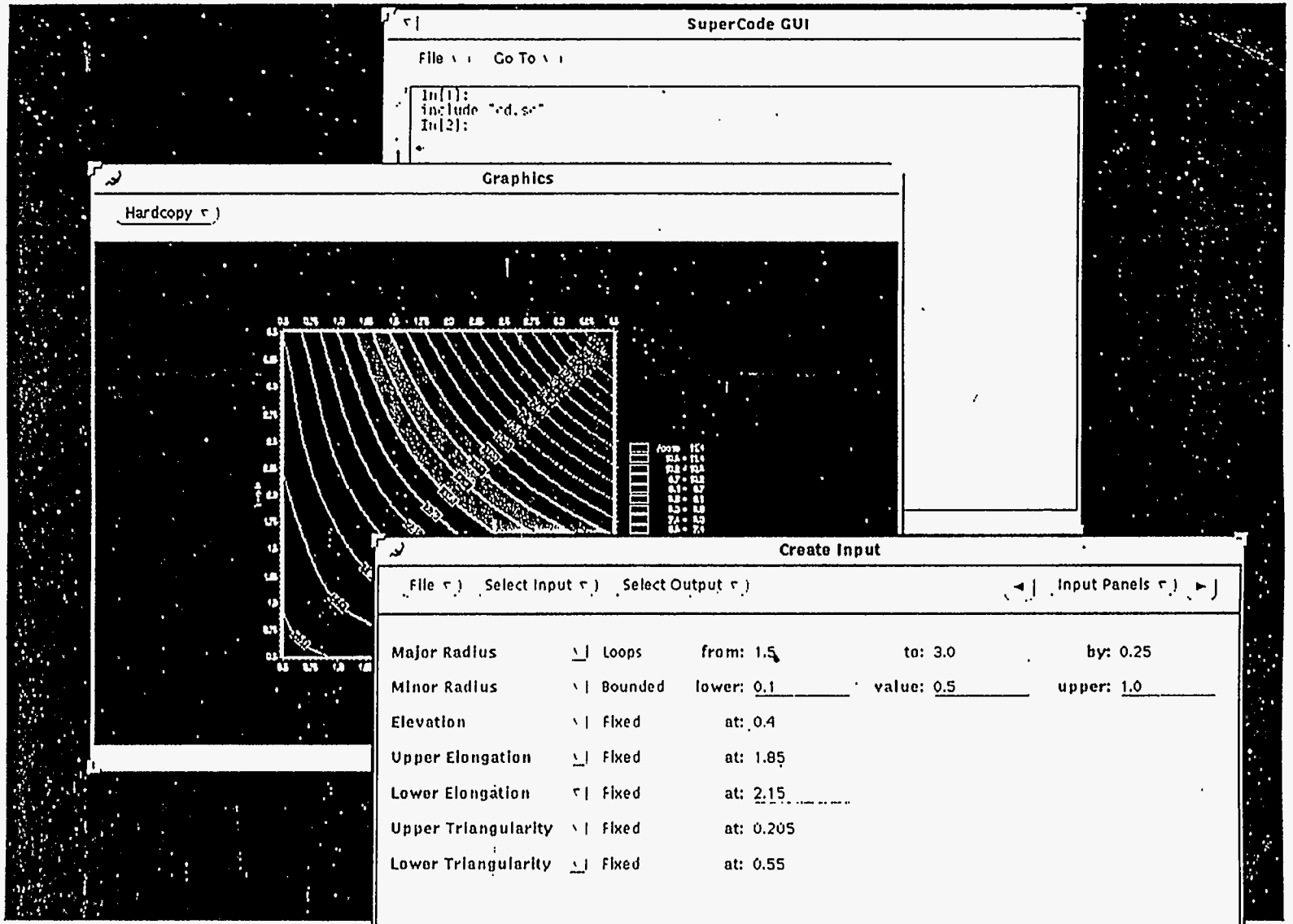
  $\Rightarrow$ All design parameters form a global variable set and are iterated simultaeously; they can be bounded if required.

  $\Rightarrow$ The detailed physics and engineering models in *SUPERCODE* are effectively function evaluators for the constraint set.

All Cases Here Were Run with *SuperCode* : Our 1-1/2D Systems and Operational Code

Features of SuperCode

- A fast, time-dependent tokamak physics and engineering simulation code
- 1D power balance with a choice of transport models (including the Rebut ∇T_c transport model)
- Simple/detailed (fast/slow) models for IC fast-wave and neutral beam current drive and/or heating.
- 2D variational equilibria
- Coupled engineering and costing models for all reactor systems.
- Constrained optimization facilities.
- Current versions on Sun, HP, IBM RS/6000 and Cray platforms.
- User-friendly (includes a programmable shell and a graphical user interface).


SuperCode GUI
File
Go To
In[1]:
In[2]:
Graphics
Hardcopy
Create Input
File
Select Input
Select Output
Input Panels
Major Radius
Loops
from: 1.5
to: 3.0
by: 0.25
Minor Radius
Bounded
lower: 0.1
value: 0.5
upper: 1.0
Elevation
Fixed
at: 0.4
Upper Elongation
Fixed
at: 1.85
Lower Elongation
Fixed
at: 2.15
Upper Triangularity
Fixed
at: 0.205
Lower Triangularity
Fixed
at: 0.55

TYPICAL EXAMPLE OF SYSTEMS CODE DESIGN OPTIMIZATION

Typical Objective:

Determine the optimum set of design variables to minimize the pilot plant direct cost[a] for a neutron wall loading of, say, 1 MW/m2.

Typical Physics and Engineering Constraints:

- Confinement: H–Factor for ITER–89–P scaling ≤ 2.0[b] (or choice of 1-D models)
- Beta: Troyon coefficient $g_T \leq 3.0$[b]
- Inductive transformer capability $\geq$ startup requirements
- Neutron wall load $\geq$ desired value (see above)
- Divertor heat loads $\leq$ specified limit[b,c]
- MHD edge safety factor, $q_{95} \geq 3.0$
- Self-consistent steady-state current $\left(J(r) = J_{BS}(r) + J_{NB}(r)\right)$ and and q profiles
- Impurity models
- ITER superconducting magnet design constraints (stress, protection, stability, etc.)
- Engineering builds and gaps, etc.
- Any other desired inequality constraints, e.g.: $Q \geq Q_{min}$, $P_{aux} \leq P_{max}$, etc.

Typical Optimization Design Variables

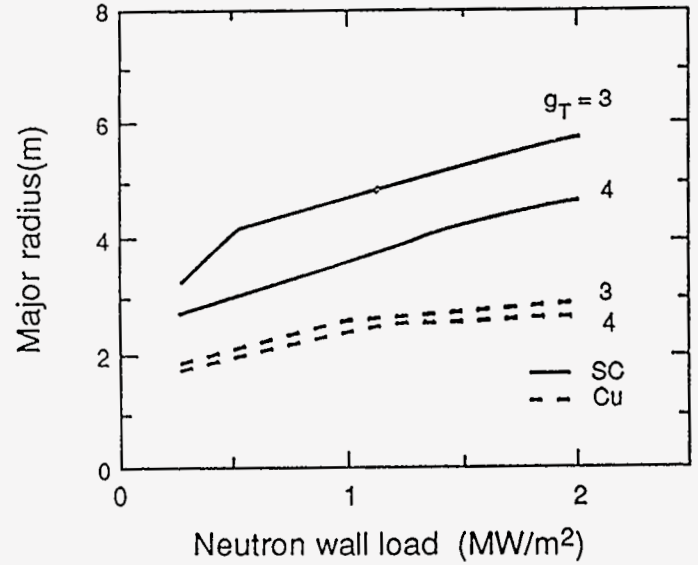
Major radius, minor radius, aspect ratio, magnetic field, plasma current, T_e, n_e, $H(\leq 2.0)$, $g_T (\leq 3.0)$ $q_{95} (\geq 3.0)$, NB injection power, NB energy, TF and OH coil geometry, vertical/radial builds, etc., ...

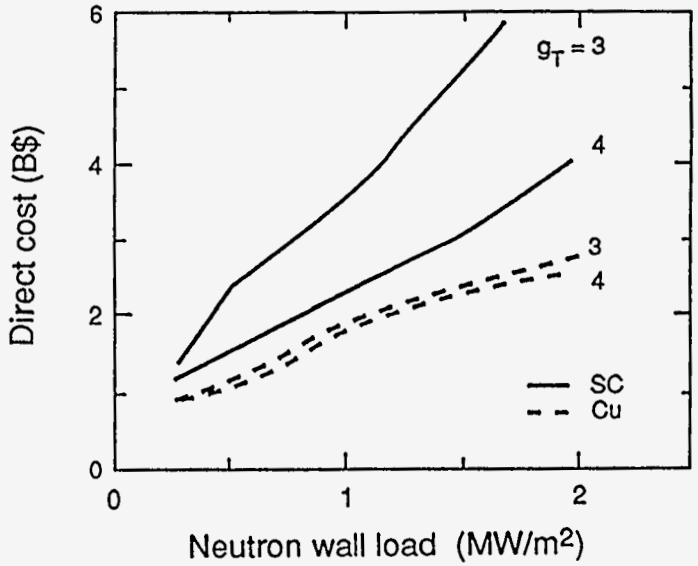
---

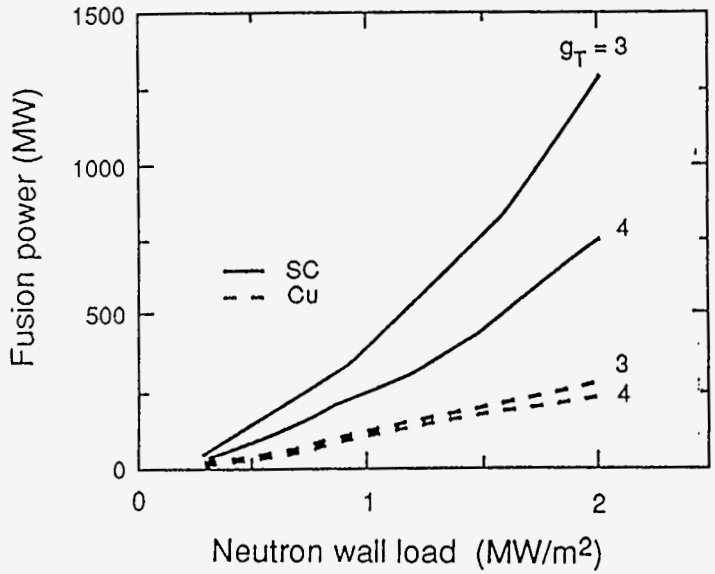
a. Note direct capital cost includes hardware and engineering. No R&D, mock-up, design, contingency, G&A, operating costs, etc., are included. Latter group can ~ double the total cost. The ITER cost database will be used initially.

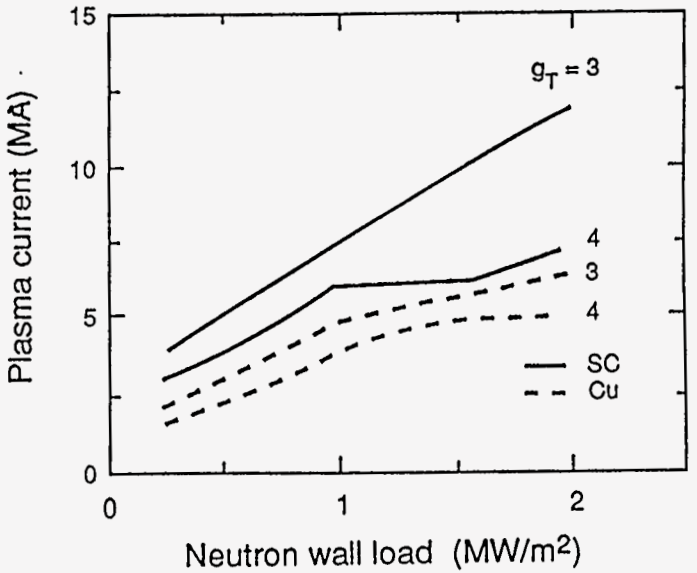
b. Important database constraints like these can be scanned as sensitivity variables.

c. Not constrained for following examples.

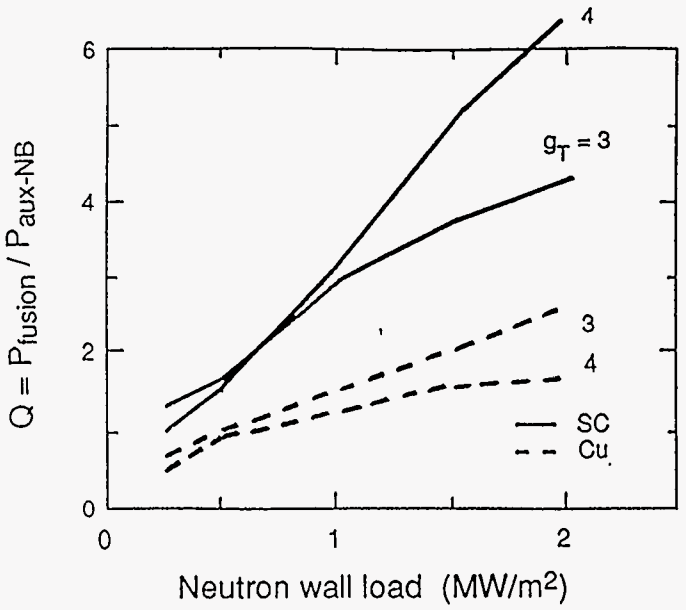


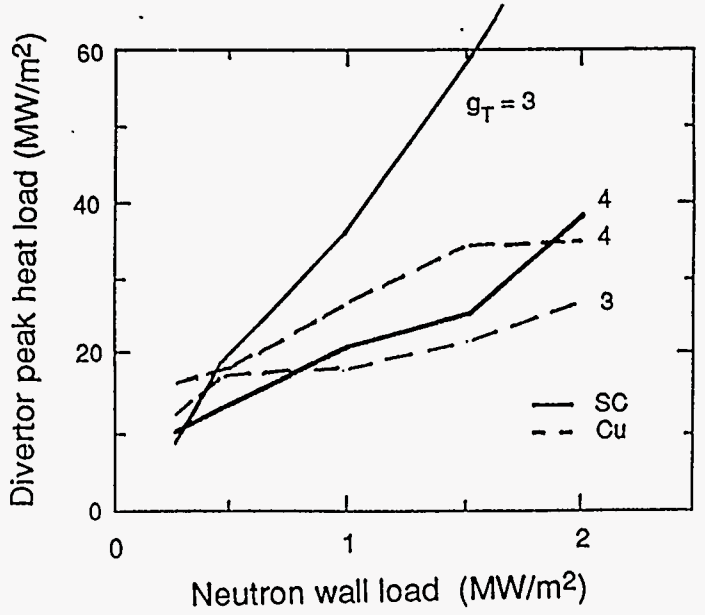


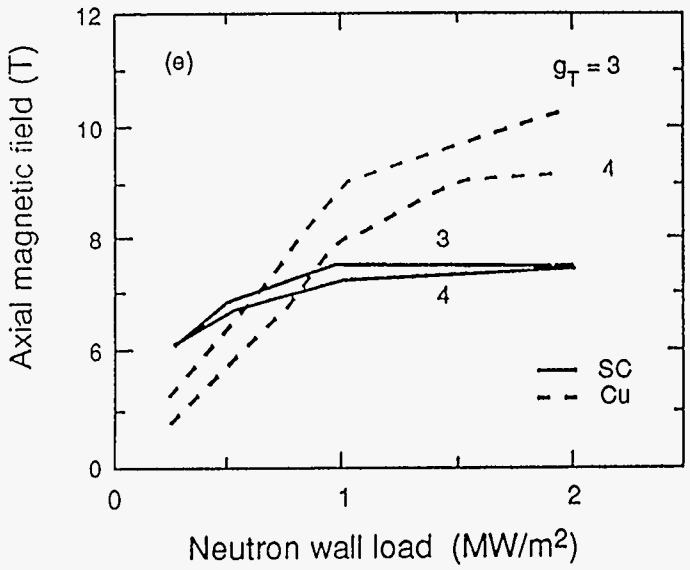


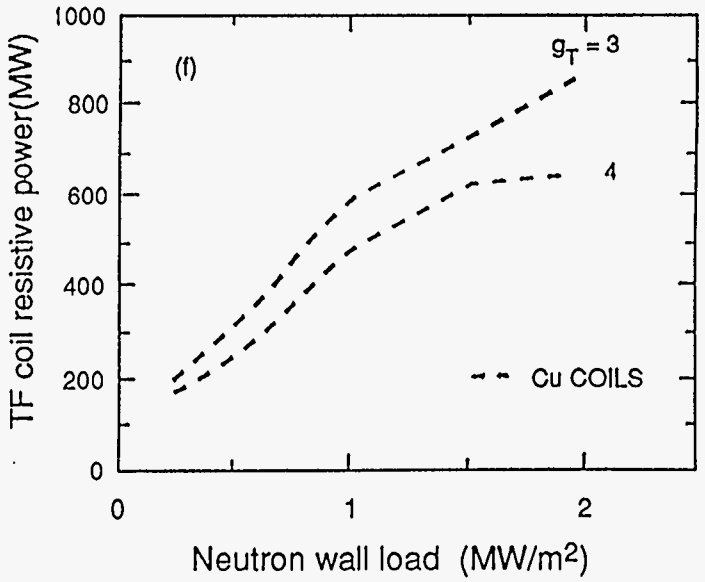


TYPICAL *SUPERCODE* RESULTS: MINIMUM COST S/C AND COPPER PILOT PLANT DESIGN POINTS -V- REQUIRED NEUTRON WALL LOADING.
(Max permitted Troyon beta coefficient g_T is shown as a parameter)

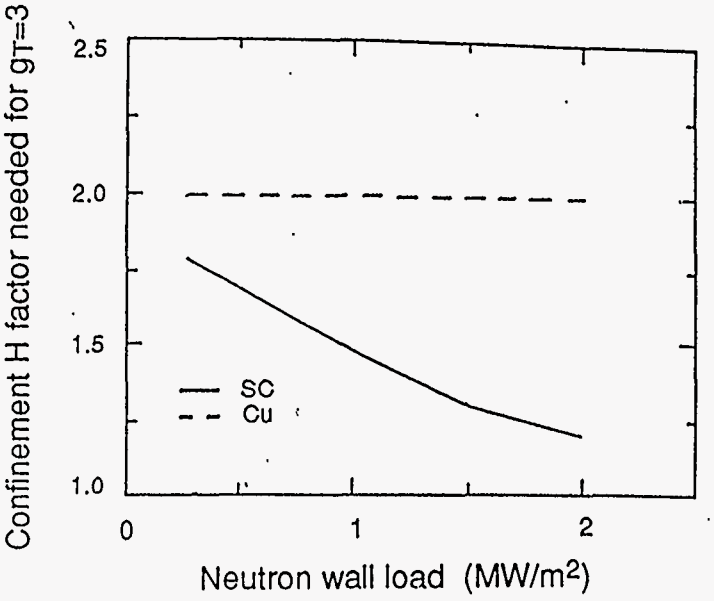








TYPICAL *SUPERCODE* RESULTS: MINIMUM COST S/C AND COPPER PILOT PLANT DESIGN POINTS -V- REQUIRED NEUTRON WALL LOADING -- continued.
(Max permitted Troyon beta coefficient g_T is shown as a parameter)



Note: beta coeff. g_T is always at its max. permitted value.

TYPICAL *SUPERCODE* RESULTS: MINIMUM COST S/C AND COPPER PILOT PLANT DESIGN POINTS -V- REQUIRED NEUTRON WALL LOADING -- continued.

TYPICAL *SUPERCODE* PILOT PLANT RESULTS: MINIMUM COST SUPERCONDUCTING DESIGNS

	Average neutron wall load	
	0.5 MW/m^2	1 MW/m^2
Direct cost (M$)	2380	3470
Power operating cost** (M$/yr)	37	62
Major radius (m)	4.14	471
Aspect ratio	5.0*	4.8
Axial magnetic field (T)	5.2	7.3
Plasma current (MA)	6.7	7.6
Total fusion power (MW)	145	394
Fusion production: thermonuclear/beam-plasma (%)	79/21	89/14
NB injection power (MW)	79	131
NB injection energy (MV)	1.0	1.4
Q	1.8	3.0
T_e (keV)	8.6	10.6
n_e (10^{20} m^{-3})	1.45	1.58
q_{95}	30*	3.0*
Confinement factor H	1.7	1.5
Troyon beta coefficient g_T	3 .0*	3.0*
Bootstrap current fraction	0.49	0.43
Divertor peak heat load (MW/m^2)	18	36
Field at TF Coil (T)	13.1	13.7

* At a constraint bound

** Electric power @ 54 mill/kWhr for 50% availability

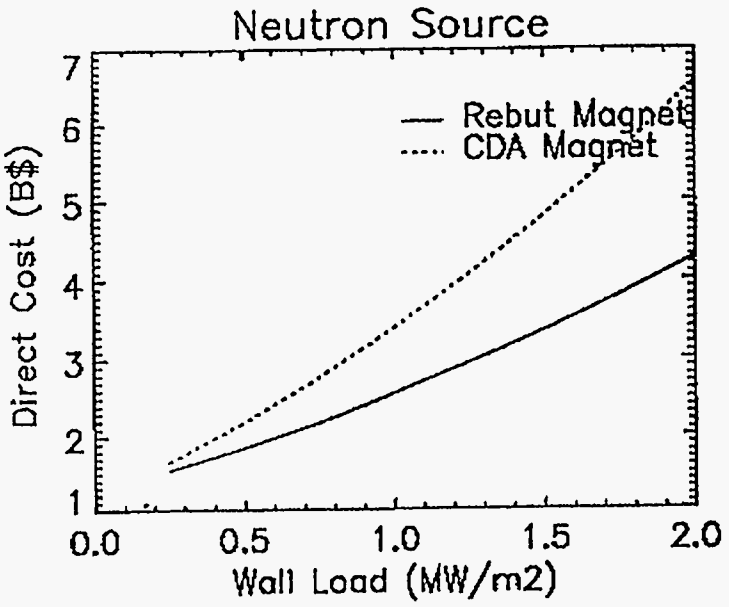
TYPICAL *SUPERCODE* PILOT PLANT RESULTS: MINIMUM COST COPPER-COIL DESIGNS

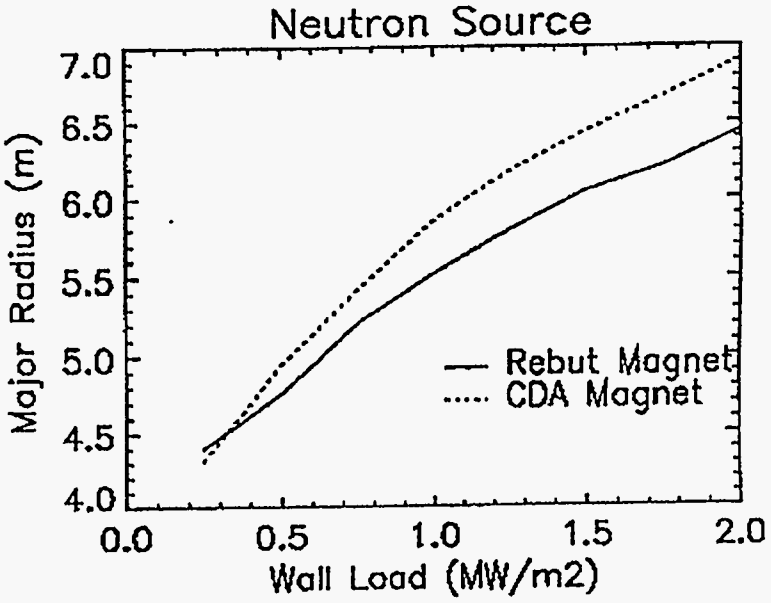
	Average neutron wall load	
	0.5 MW/m^2	1 MW/m^2
Direct cost (M$)	1140	18 80
Power operating cost** (M$/yr)	93	178
Major radius (m)	2.12	2.60
Aspect ratio	5.0*	5.0*
Axial magnetic field (T)	6.5	8.9
Plasma current (MA)	2.8	4.7
Total fusion power (MW)	38	115
Fusion production: thermonuclear/beam-plasma (%)	18/82	73/27
NB injection power (MW)	40	77
NB injection energy (MV)	0.23	0.98
Q	0.94	1.5
T_e (KeV)	8.4	9.2
n_e (10^{20} m^{-3})	1.12	2.32
q_{95}	3.0*	3.0*
Confinement factor H	2.0*	2.0*
Troyon beta coefficient g_T	3.0*	3.0*
Bootstrap current fraction	0.44	0.45
Divertor peak heat load (MW/m^2)	17	18
Field at TF Coil (T)	11.0	14.6
TF resistive power (MW)	315	600

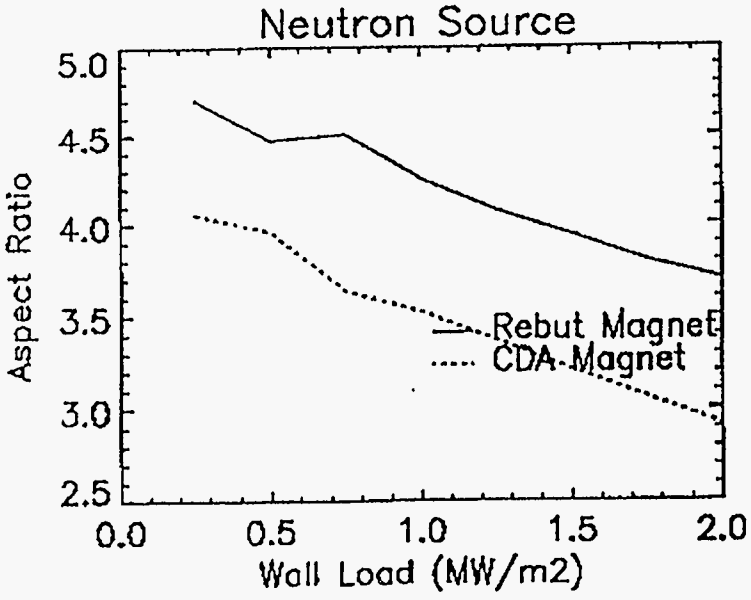
* At a constraint bound

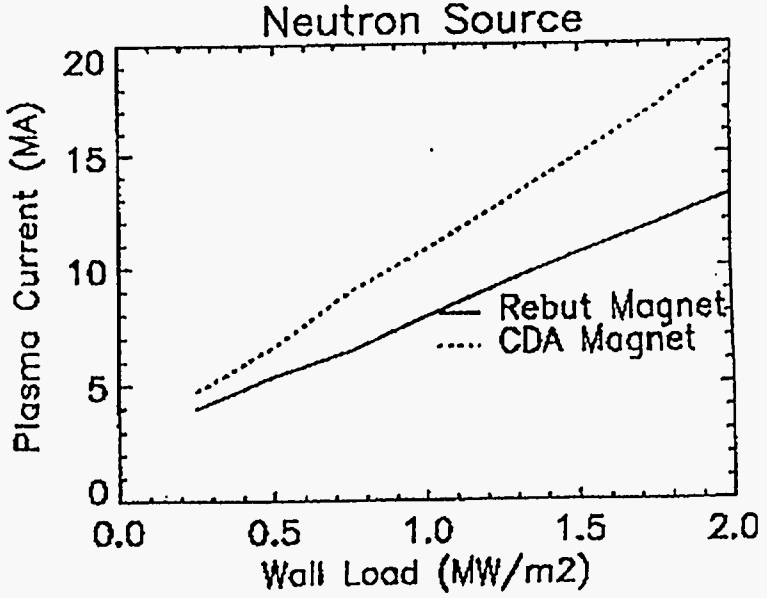
** Electric power @ 54 mill/kWhr for 50% availability

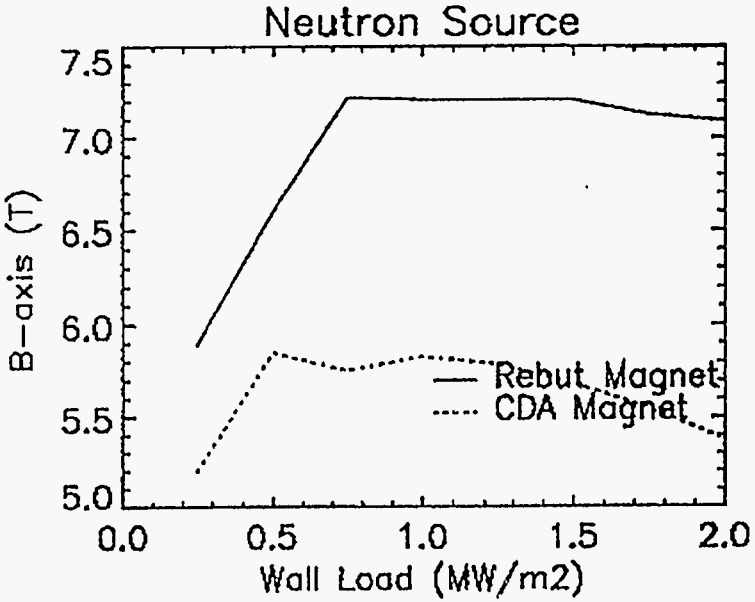
PARAMETER	NOMINAL VALUE ({}= upbuttons for min size S/C machine)	SENSITIVITIES
Neutron wall load (MW/m2)	1.0 {scan}	0.25 - 2.0
Troyon coeff (%)	≤3.0 {3.5}	2.5 - 3.5
H (ITER-89-P)	≤2.0	1.5 - 2.5
Elongation @ 95%	≤1.6 {1.8}	1.4 - 2.0
i.b scrapeoff (cm)	15 {10}	7-20
i.b build	CDA + 10cm = EDA less10cm {minimum thickness w/tungsten}	to be discussed
TF magnet config.	S/C EDA-like	(1)S/C CDA-like. (2)copper alloy
q @ 95%	≤3.0	
V-sec	Startup to full current	Fraction of full startup
Divertor heatload	Unconstrained	~5-20MW/m2 with and without impurity seeding
Alpha ash fraction	3%	>3%
Impurities	C, O, Fe as per CDA $=f(n_e)$	1% Be
CD method	Neutral beams	ICFW
Minimum minor radius	≥0.5m	
Maximum aspect ratio	≤5.0	

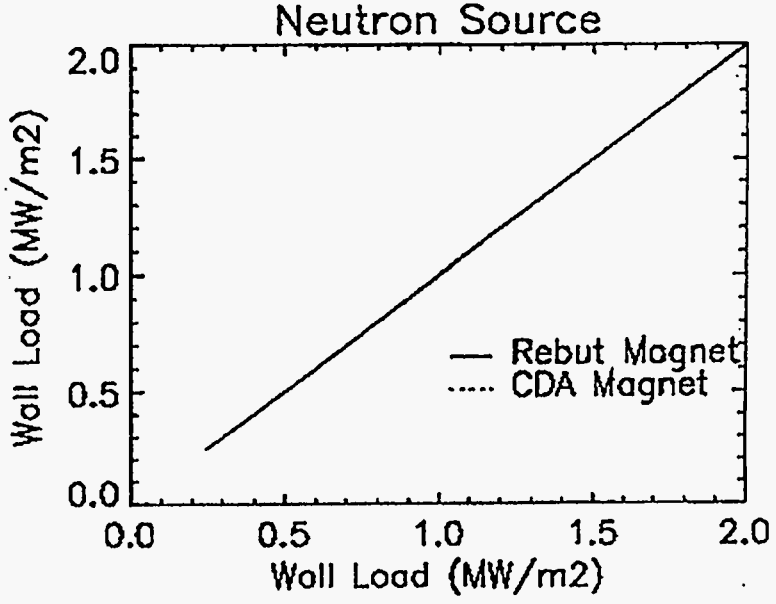

Neutron Source
Direct Cost (B$)
Wall Load (MW/m2)
Rebut Magnet
CDA Magnet
1
2
3
4
5
6
7
0.0
0.5
1.0
1.5
2.0

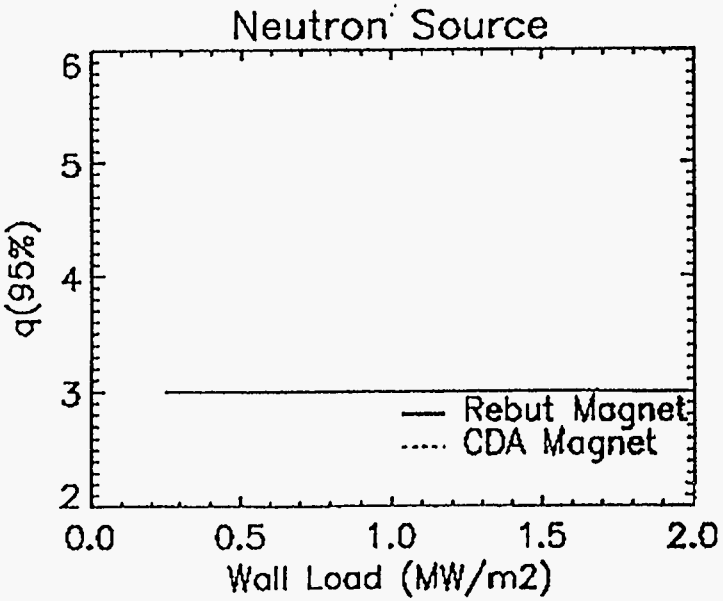

Neutron Source
Major Radius (m)
Wall Load (MW/m2)
Rebut Magnet
CDA Magnet
4.0
4.5
5.0
5.5
6.0
6.5
7.0
0.0
0.5
1.0
1.5
2.0

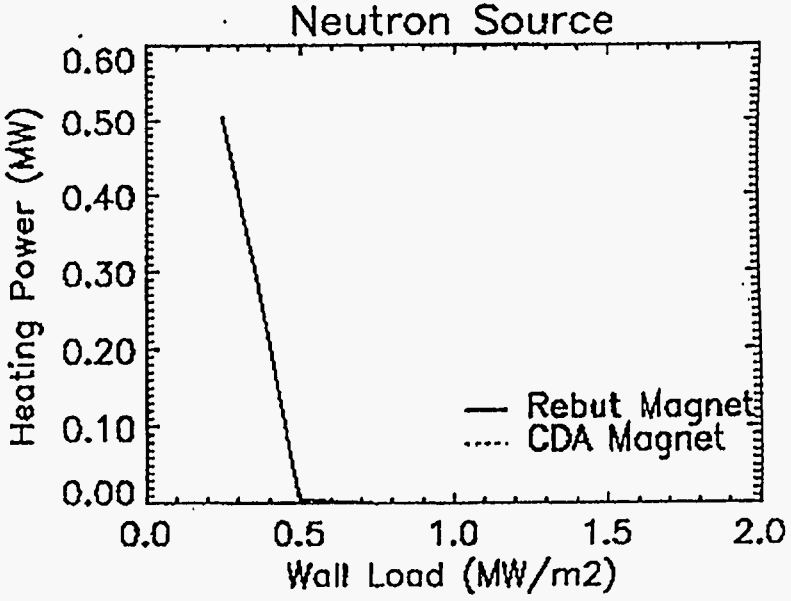

Neutron Source
Aspect Ratio
Wall Load (MW/m2)
Rebut Magnet
CDA Magnet
2.5
3.0
3.5
4.0
4.5
5.0
0.0
0.5
1.0
1.5
2.0

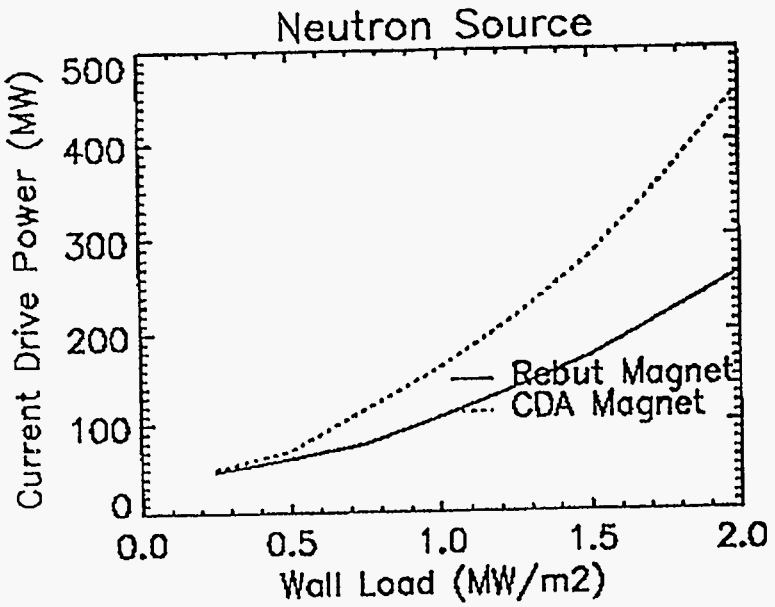

Neutron Source
Plasma Current (MA)
Wall Load (MW/m2)
Rebut Magnet
CDA Magnet
0
5
10
15
20
0.0
0.5
1.0
1.5
2.0

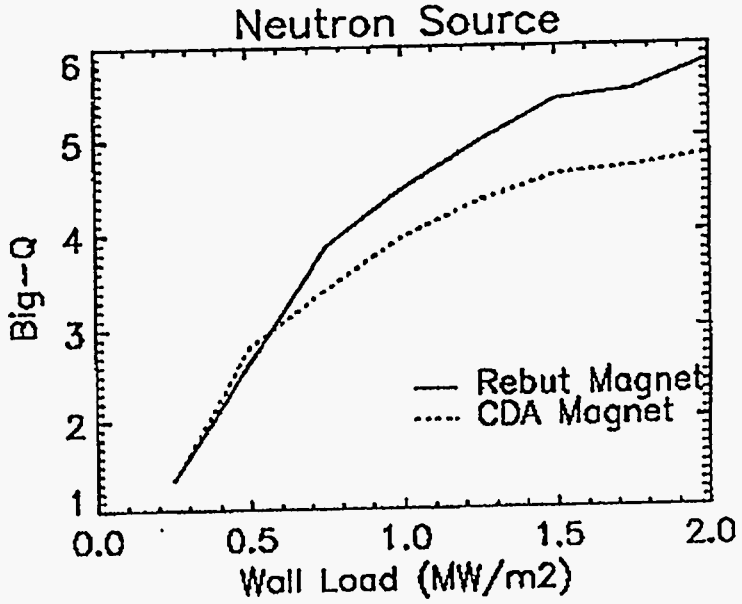

Neutron Source
B-axis (T)
7.5
7.0
6.5
6.0
5.5
5.0
0.0
0.5
1.0
1.5
2.0
Wall Load (MW/m2)
Rebut Magnet
CDA Magnet

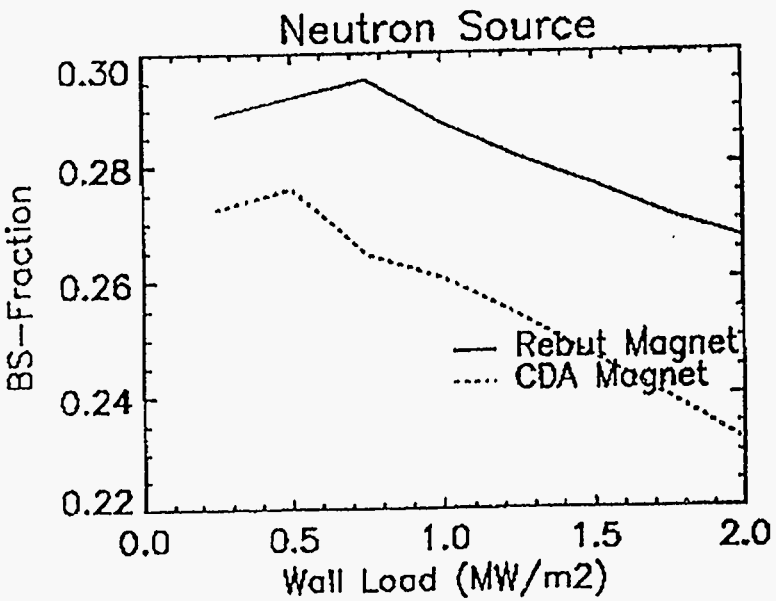

Neutron Source
Wall Load (MW/m2)
2.0
1.5
1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
Wall Load (MW/m2)
Rebut Magnet
CDA Magnet

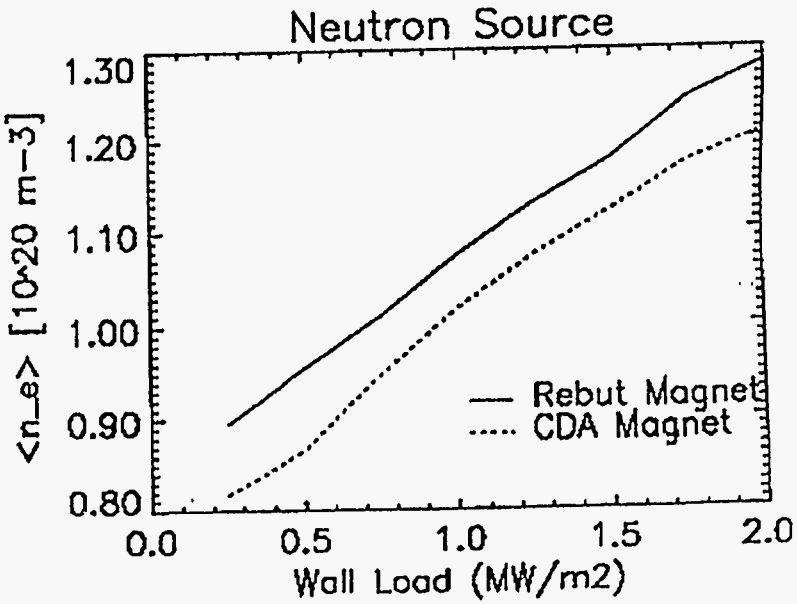

Neutron Source
q(95%)
6
5
4
3
2
0.0
0.5
1.0
1.5
2.0
Wall Load (MW/m2)
Rebut Magnet
CDA Magnet

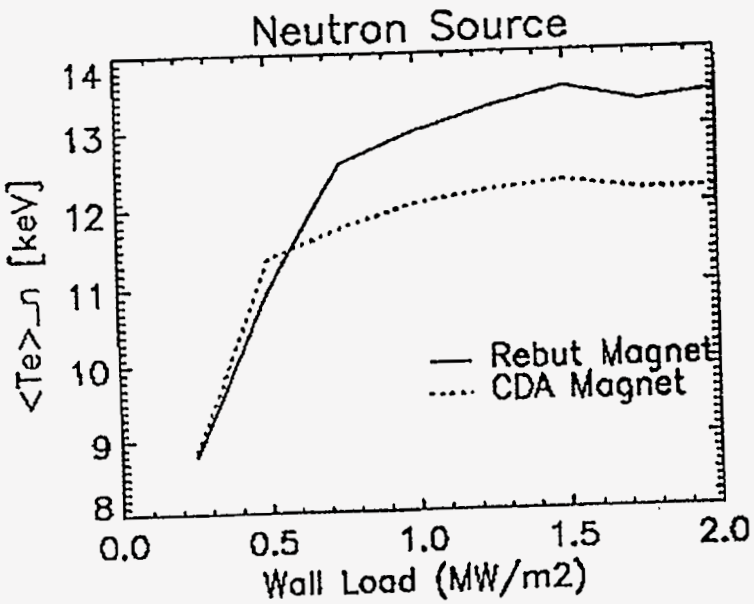

Neutron Source
Heating Power (MW)
0.60
0.50
0.40
0.30
0.20
0.10
0.00
0.0
0.5
1.0
1.5
2.0
Wall Load (MW/m2)
Rebut Magnet
CDA Magnet

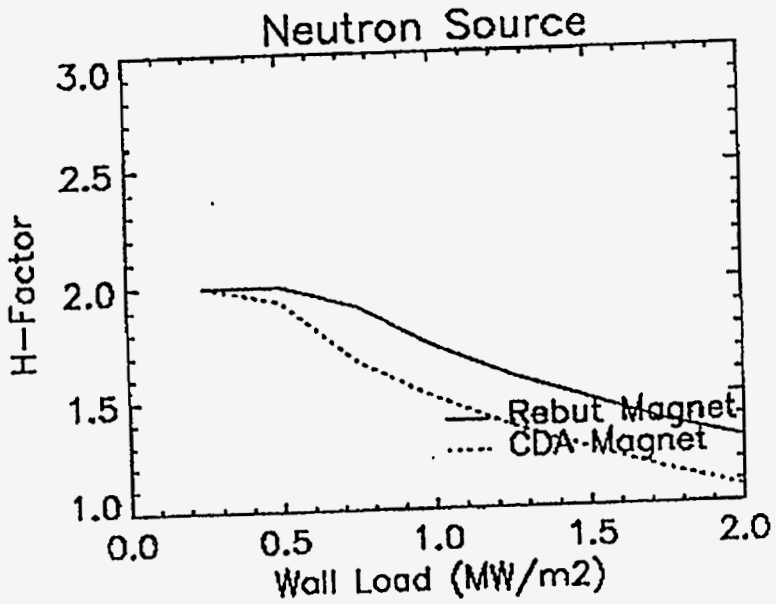

Neutron Source
Current Drive Power (MW)
500
400
300
200
100
0
0.0
0.5
1.0
1.5
2.0
Wall Load (MW/m2)
Rebut Magnet
CDA Magnet

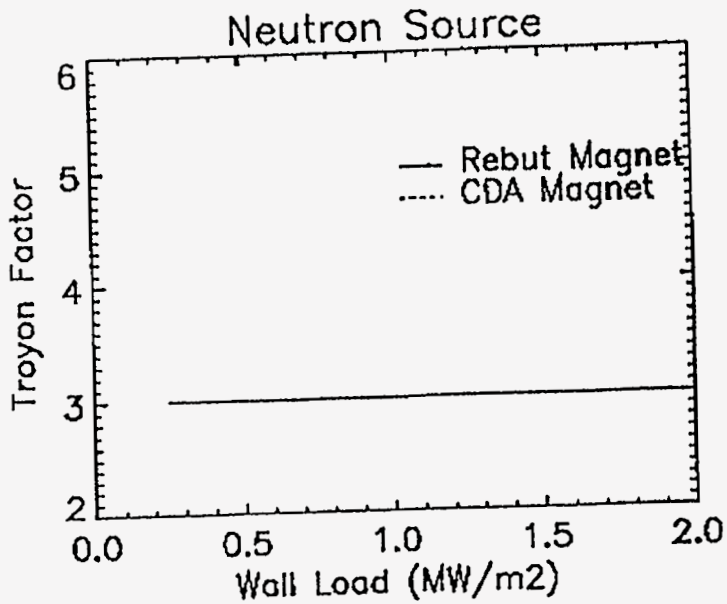

Neutron Source
Big-Q
6
5
4
3
2
1
0.0
0.5
1.0
1.5
2.0
Wall Load (MW/m2)
Rebut Magnet
CDA Magnet

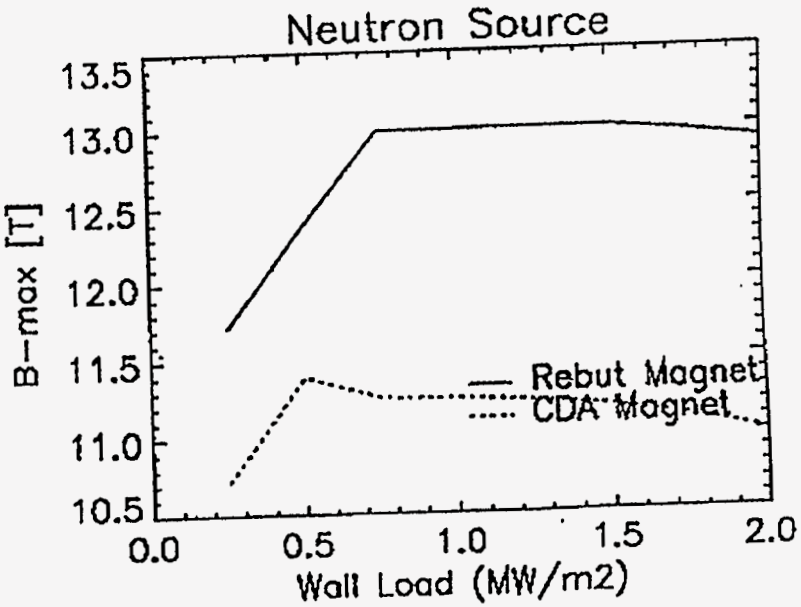

Neutron Source
BS-Fraction
0.30
0.28
0.26
0.24
0.22
0.0
0.5
1.0
1.5
2.0
Wall Load (MW/m2)
Rebut Magnet
CDA Magnet

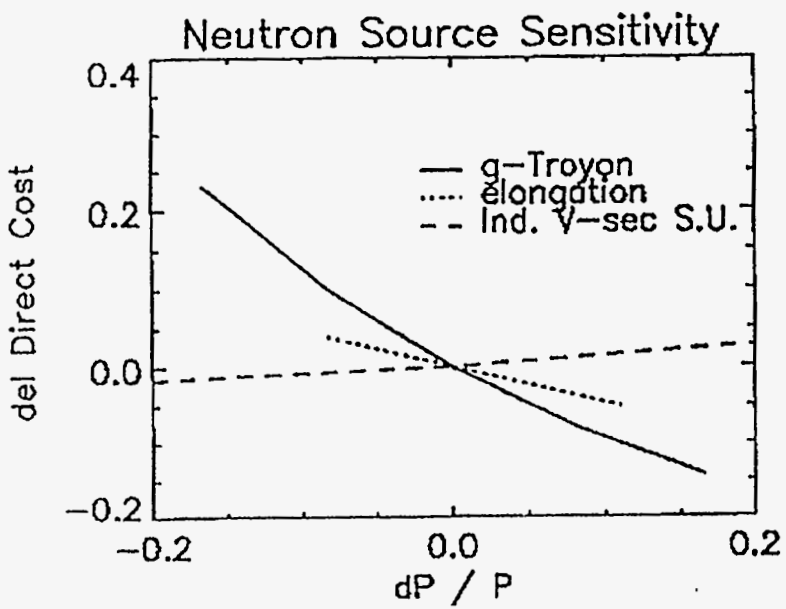

Neutron Source
<n_e> [10^20 m-3]
1.30
1.20
1.10
1.00
0.90
0.80
0.0
0.5
1.0
1.5
2.0
Wall Load (MW/m2)
Rebut Magnet
CDA Magnet

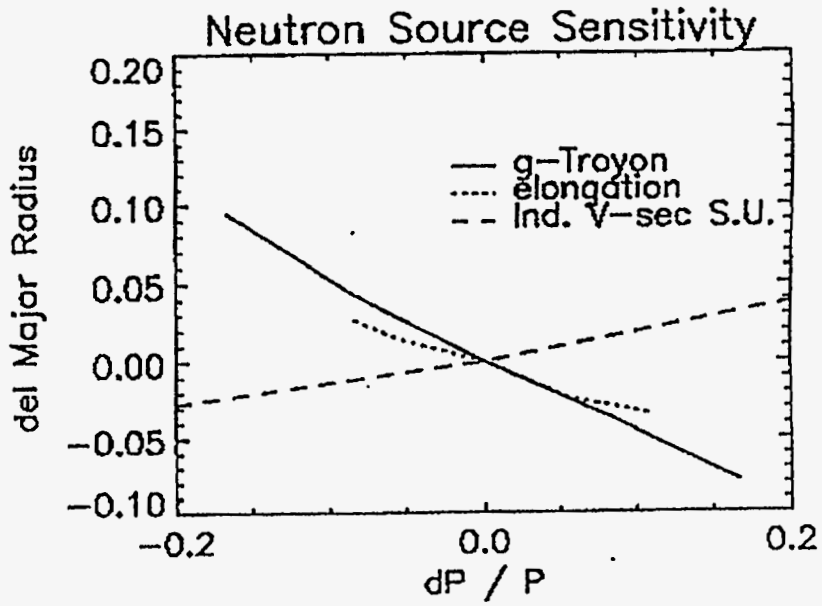

Neutron Source
<Te>_n [keV]
14
13
12
11
10
9
8
0.0
0.5
1.0
1.5
2.0
Wall Load (MW/m2)
Rebut Magnet
CDA Magnet

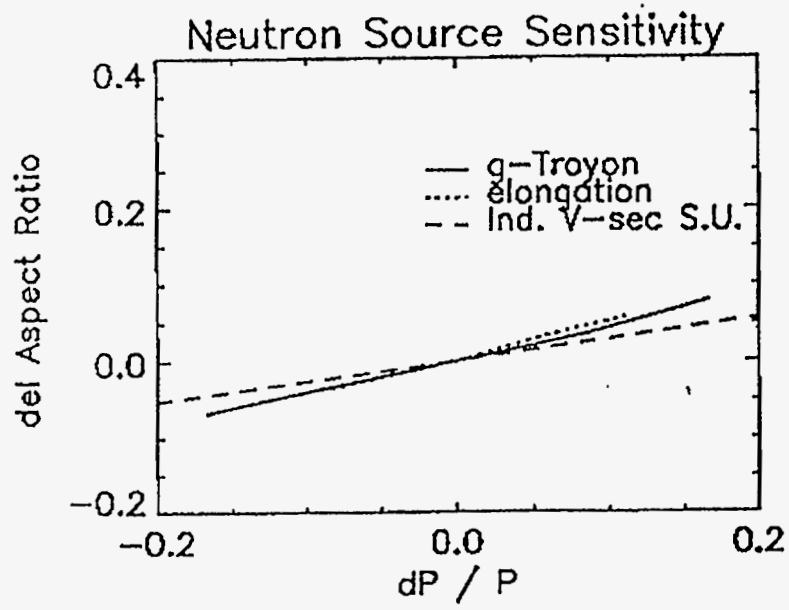

Neutron Source
H-Factor
3.0
2.5
2.0
1.5
1.0
0.0
0.5
1.0
1.5
2.0
Wall Load (MW/m2)
Rebut Magnet
CDA Magnet

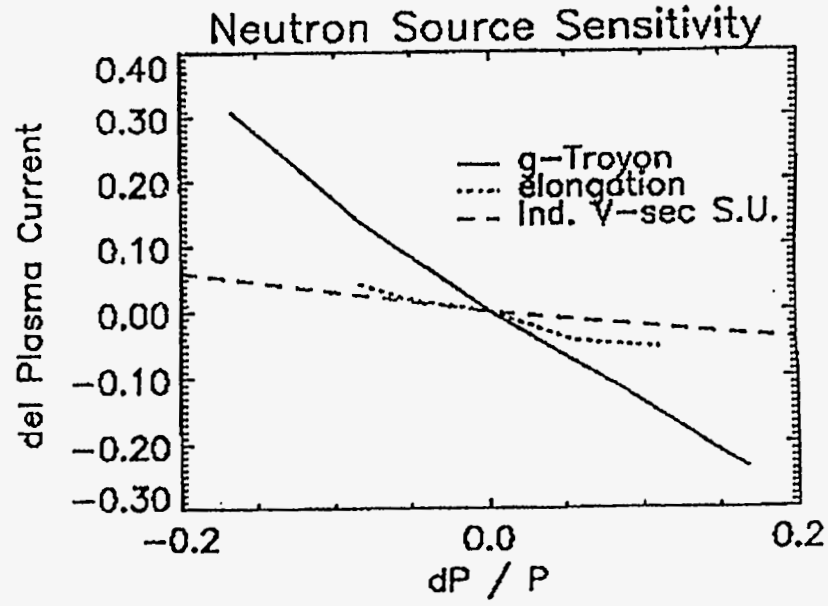

Neutron Source
Troyon Factor
6
5
4
3
2
0.0
0.5
1.0
1.5
2.0
Wall Load (MW/m2)
Rebut Magnet
CDA Magnet

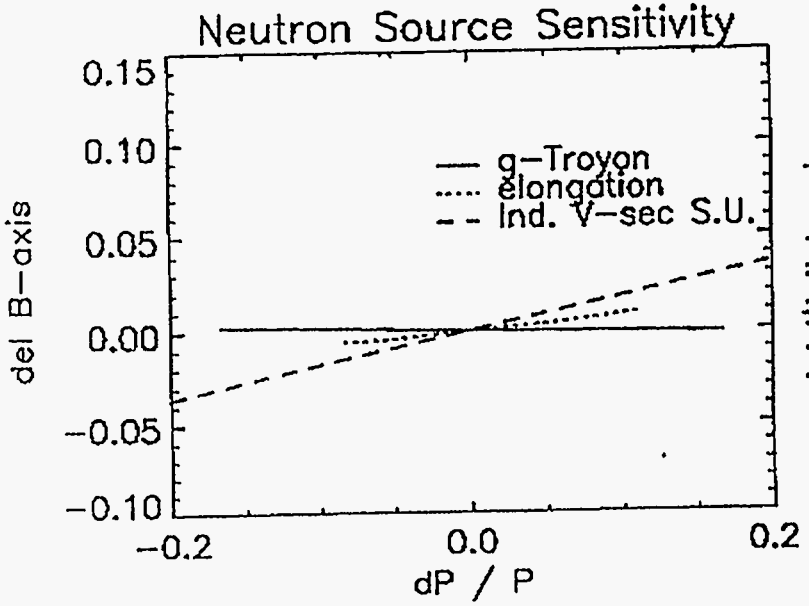

Neutron Source
B-max [T]
13.5
13.0
12.5
12.0
11.5
11.0
10.5
0.0
0.5
1.0
1.5
2.0
Wall Load (MW/m2)
Rebut Magnet
CDA Magnet

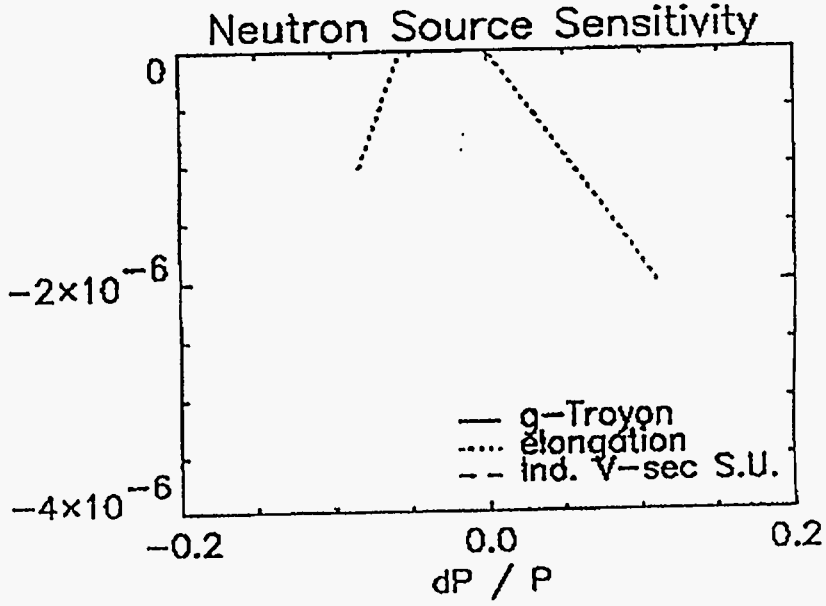

Neutron Source Sensitivity
del Direct Cost
dP / P
g–Troyon
elongation
Ind. V–sec S.U.
0.4
0.2
0.0
−0.2
−0.2
0.0
0.2

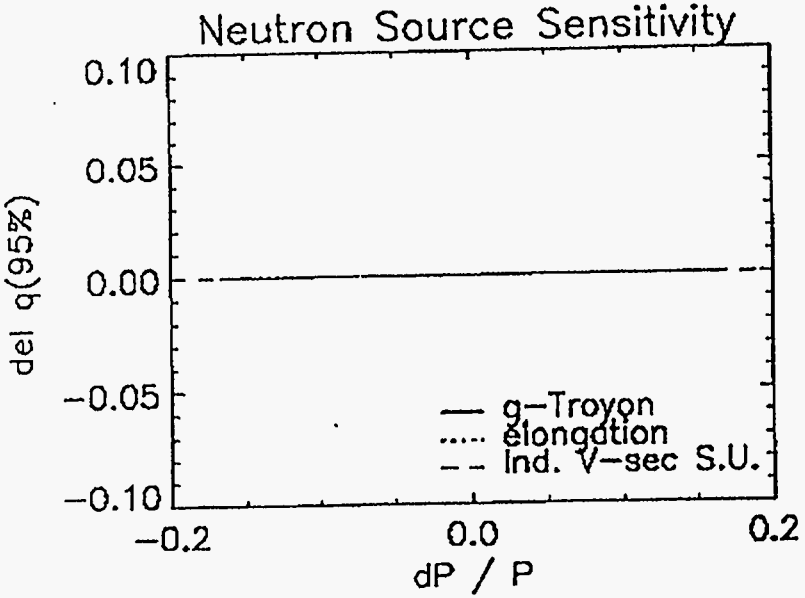

Neutron Source Sensitivity
del Major Radius
dP / P
g–Troyon
elongation
Ind. V–sec S.U.
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.2
0.0
0.2

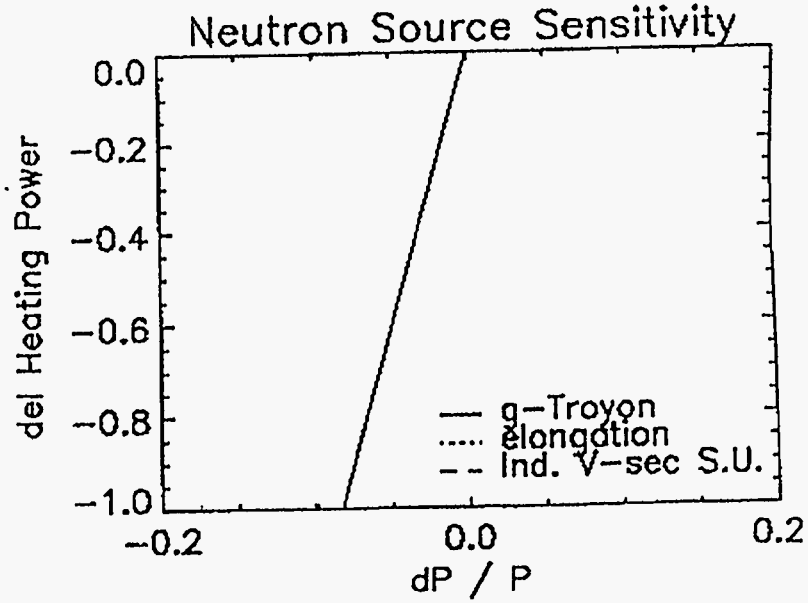

Neutron Source Sensitivity
del Aspect Ratio
dP / P
g–Troyon
elongation
Ind. V–sec S.U.
0.4
0.2
0.0
−0.2
−0.2
0.0
0.2

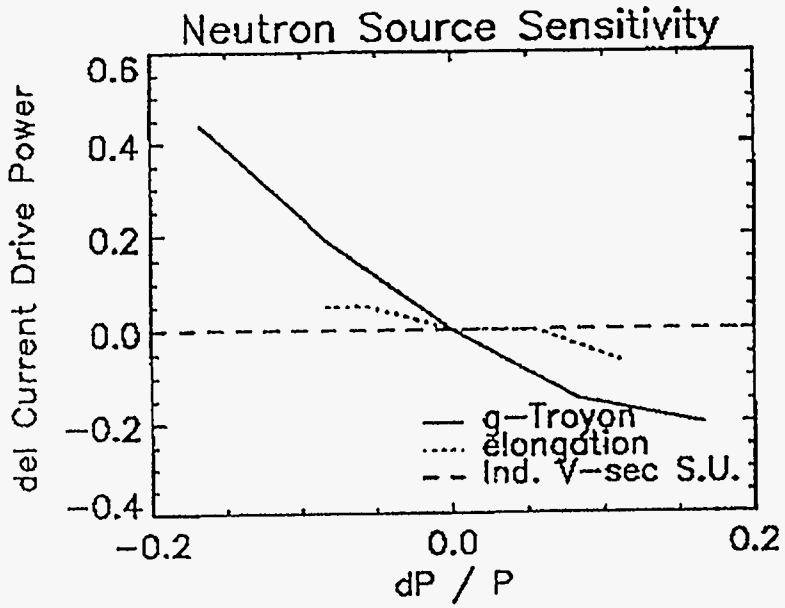

Neutron Source Sensitivity
del Plasma Current
dP / P
g–Troyon
elongation
Ind. V–sec S.U.
0.40
0.30
0.20
0.10
0.00
−0.10
−0.20
−0.30
−0.2
0.0
0.2

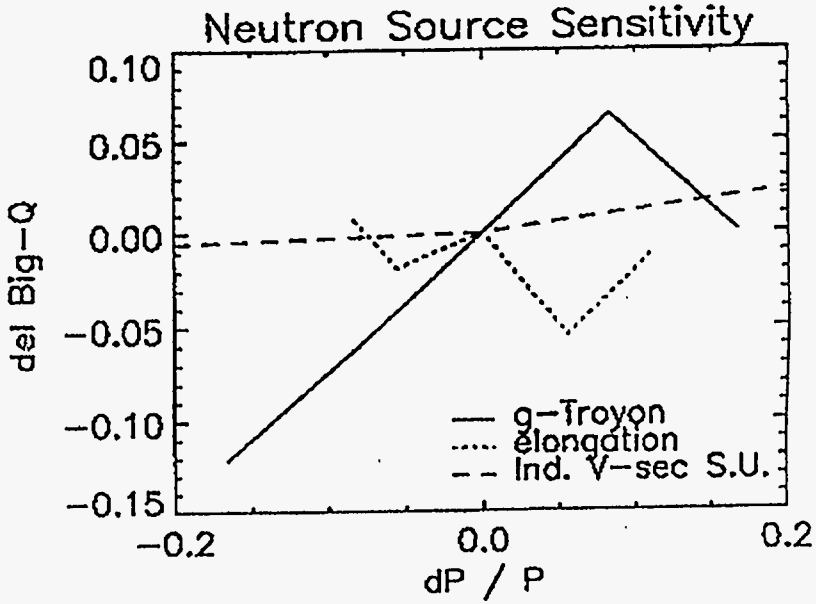

Neutron Source Sensitivity
del B-axis
0.15
0.10
0.05
0.00
-0.05
-0.10
-0.2
0.0
0.2
dP / P
g-Troyon
elongation
Ind. V-sec S.U.

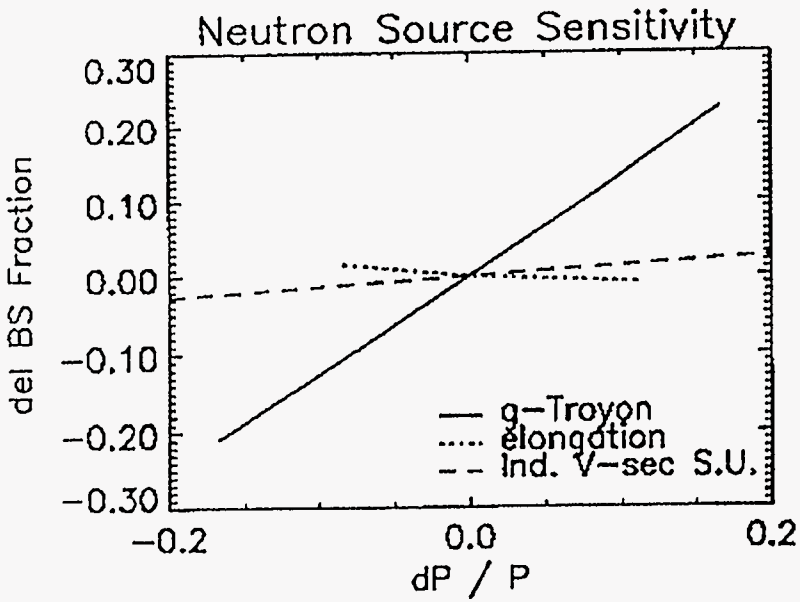

Neutron Source Sensitivity
del Wall Load
0
-2x10^-6
-4x10^-6
-0.2
0.0
0.2
dP / P
g-Troyon
elongation
Ind. V-sec S.U.

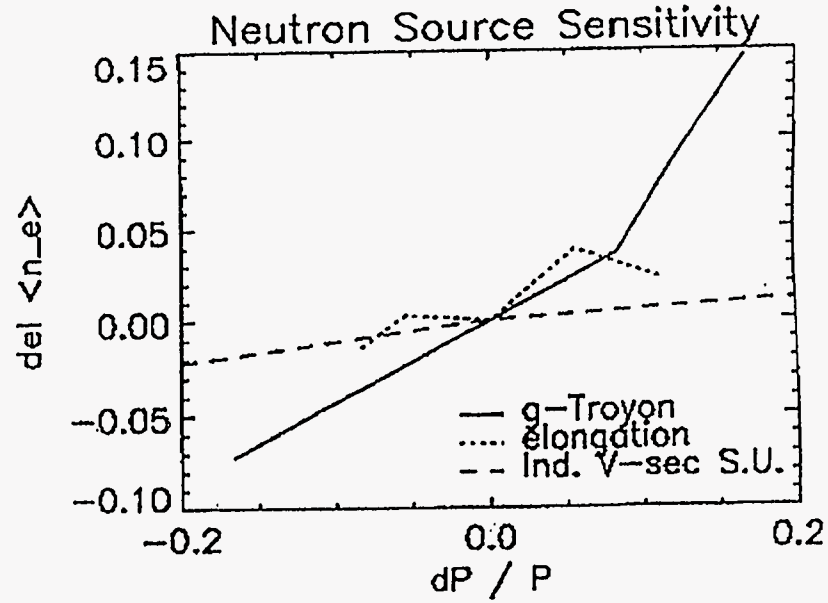

Neutron Source Sensitivity
del q(95%)
0.10
0.05
0.00
-0.05
-0.10
-0.2
0.0
0.2
dP / P
g-Troyon
elongation
Ind. V-sec S.U.

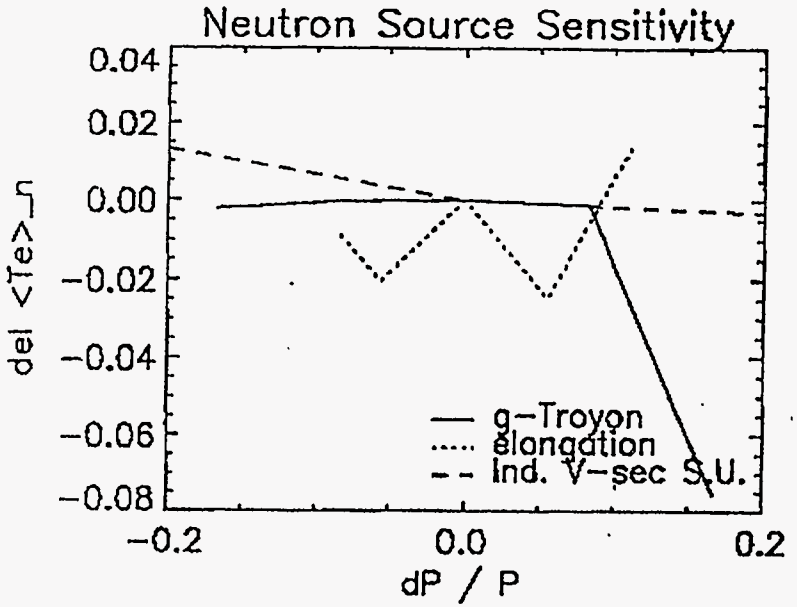

Neutron Source Sensitivity
del Heating Power
0.0
-0.2
-0.4
-0.6
-0.8
-1.0
-0.2
0.0
0.2
dP / P
g-Troyon
elongation
Ind. V-sec S.U.

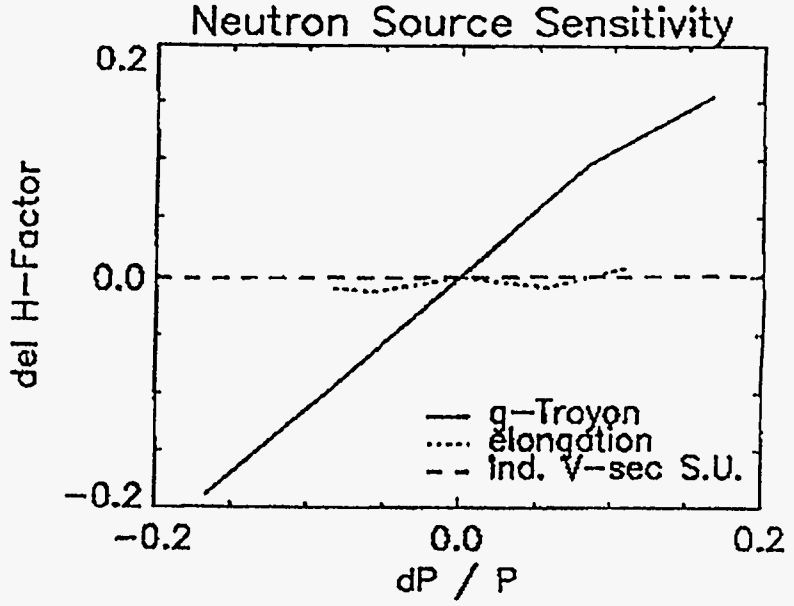

Neutron Source Sensitivity
del Current Drive Power
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.2
0.0
0.2
dP / P
g−Troyon
elongation
Ind. V−sec S.U.


Neutron Source Sensitivity
del Big−Q
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.2
0.0
0.2
dP / P
g−Troyon
elongation
Ind. V−sec S.U.

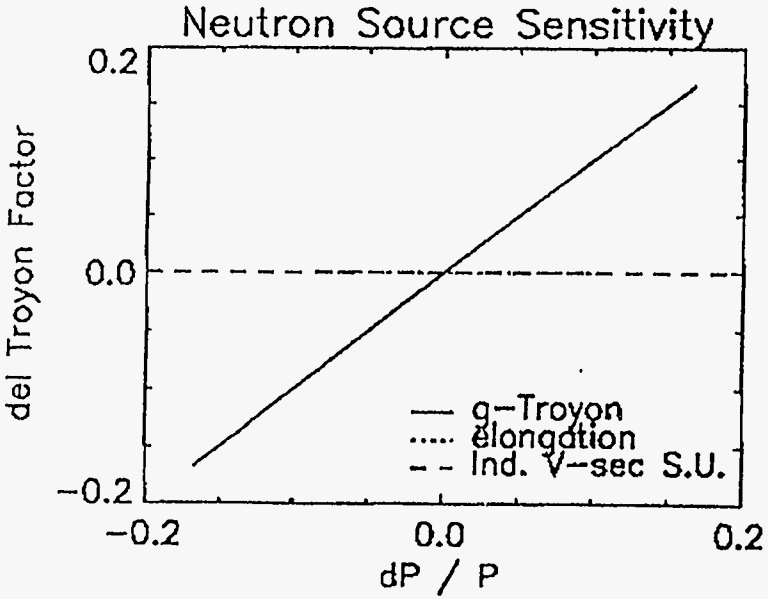

Neutron Source Sensitivity
del BS Fraction
0.30
0.20
0.10
0.00
−0.10
−0.20
−0.30
−0.2
0.0
0.2
dP / P
g−Troyon
elongation
Ind. V−sec S.U.

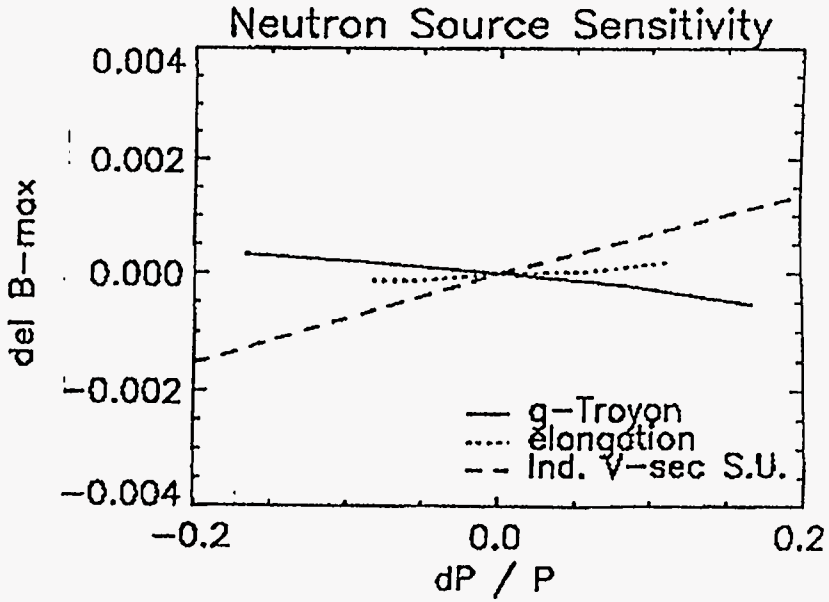

Neutron Source Sensitivity
del <n_e>
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.2
0.0
0.2
dP / P
g−Troyon
elongation
Ind. V−sec S.U.


Neutron Source Sensitivity
del <Te>_n
0.04
0.02
0.00
−0.02
−0.04
−0.06
−0.08
−0.2
0.0
0.2
dP / P
g−Troyon
elongation
Ind. V−sec S.U.


Neutron Source Sensitivity
del H−Factor
0.2
0.0
−0.2
−0.2
0.0
0.2
dP / P
g−Troyon
elongation
Ind. V−sec S.U.

16


Neutron Source Sensitivity
del Troyon Factor
0.2
0.0
−0.2
−0.2
0.0
0.2
dP / P
g−Troyon
elongation
Ind. V−sec S.U.


Neutron Source Sensitivity
del B−max
0.004
0.002
0.000
−0.002
−0.004
−0.2
0.0
0.2
dP / P
g−Troyon
elongation
Ind. V−sec S.U.

Our Recommendations

- Acknowledge the need to establish continuing research into new and substantially different fusion concepts. We're not satisfied with our present, conventional approaches!

- Earmark funds for conceptual development and computational modeling, to the stage where new, definitive experiments can be defined.

- Promote intellectual stimulation in breadth, by encouraging high risk, high payoff approaches with the acknowledgment that only a very few may ultimately be successful.

- Couple the prospective advanced physics with an engineering realization to clearly identify the potential for a step change in capital costs, complexity and development path relative to our present concepts. (e.g, Assume the advanced physics works -- how does it make a better reactor?)

OBJECTIVE

Demonstrate the value-of-information of the physics on commercial tokamak reactors.

Where does the benefit of advanced physics saturate?

How?

- For the five commercial fusion reactor types shown, use the SUPERCODE Systems Code to optimize all design parameters (*major radius, aspect ratio, plasma current, field, . . ., etc.)* to minimize the cost-of-electricity (mills/kWhr), for a given net electric output.

- Scan the allowed upper limits on confinement ($1 \leq H \leq 4$) and beta ($2 \leq \beta_n \leq 8$) as sensitivity parameters* but apply the same conventional engineering constraints (shielding, magnet allowables, wall loadings, etc.) to all cases.

---

*H = rato of $\tau_E/\tau_{E(ITER\ 89\text{-}P)}$; β_n = normalized beta Troyon coefficient = $\beta(aB/I)$

The Five Types of Commercial DT Fusion Reactors

1. Pulsed Inductive Tokamak (ITER-Extrapolation):

- Ignited • Finite Q if advantageous • $q_{95} \geq 3.0$
- Pulsed, 1-10 hr inductive burn length (sensitivity shown)
- Aspect ratio optimized • TPX-like shape: $k_{95} = 2.0$, δ_{95} = optimized, DN

2. Steady-StateTokamak (TPX-Extrapolation):

- Steady-state, current-driven • $q_{95} \geq 4.0$
- Normal CD performance ($3 \times \eta_{CD}$ sensitivity) • Inductive startup only

3. Ultimate #1: The Neoclassical Tokamak Reactor:

- Neoclassical energy and alpha confinement
- $\beta \leq$ unity • Conventional ("classical") steady-state current-drive*

4. Ultimate #2: The Magnetic Toroidal Reactor:

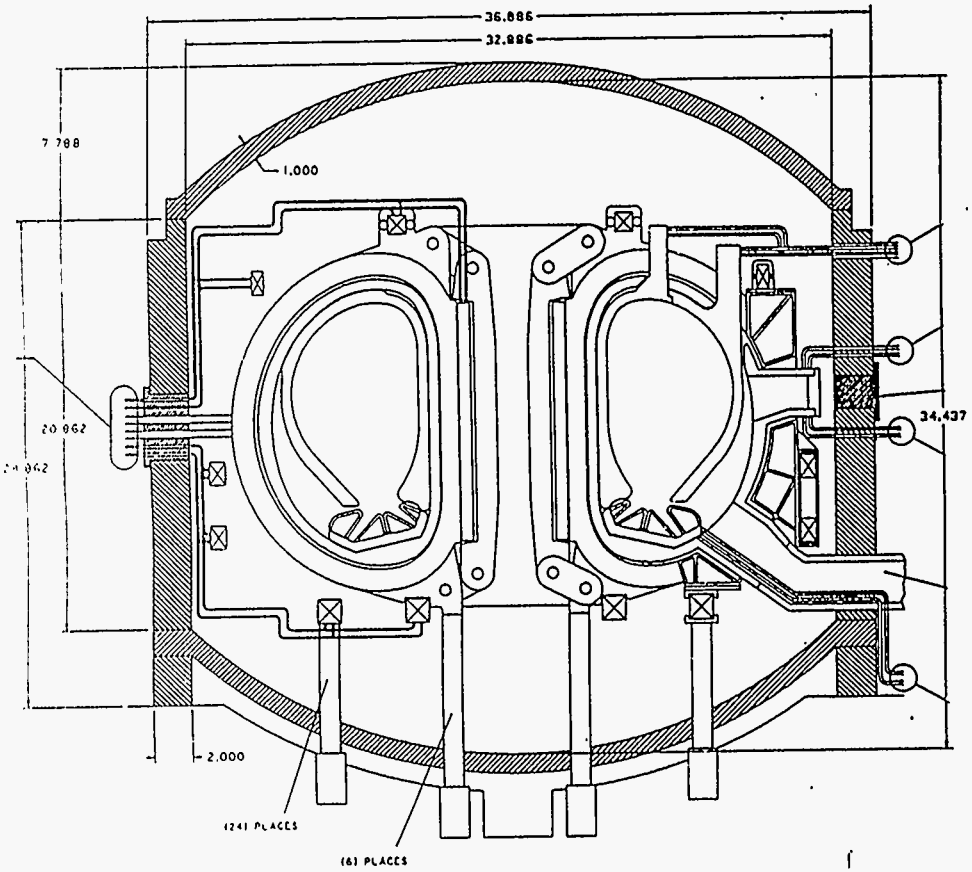
- No plasma current (not a tokamak)
- No confinement constraints • $\beta \leq$ unity

5. Ultimate #3: The Toroidal/Point Source Reactor:

- No magnetic field • Can be toroidal or, equivalently, a point source
- Also equivalent to the ultimate ICF reactor, i.e no driver cost and infinite rep-rate

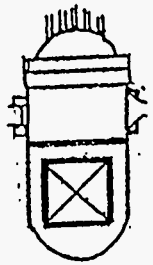
* Although typically very high bootstrap fraction (>90%)

The Competition :



International Thermonuclear Experimental Reactor (ITER) EDA

Volume:	25,600m^3	-v-	167m^3	*(factor of 154)*
Mass:	40,560tn	-v-	630tn	*(factor of 64)*
Cost:	$3137M	-v-	$46M, w/o fuel	*(factor of 68)*
			$106M, w/ fuel	*(factor of 30)*



Westinghouse AP-600:
Advanced, Passively-safe, LWR

THE COMPETITION

COMPARISON OF FISSION AND FUSION CAPITAL COSTS ($-1992)

(Values in parentheses are relative to fission)

	BETTER EXPERIENCE PWR [a] (M$1992)	ARIES-I' [b] (M$1992)	ITER EDA (8/93) [c] (M$1992)
LAND AND STRUCTURES	283 *(1.0)*	255 *(0.90)*	787 *(2.8)*
Reactor building and hot cells	88 *(1.0)*	140 *(1.6)*	464 *(5.3)*
Other structures and land	195 *(1.0)*	115 *(0.59*	323[g] *(1.7)*
REACTOR PLANT EQUIPMENT	382 *(1.0)*	1683 *(4.4)*	4706 *(12.0)*
Nuclear Island / Fusion power core [d]	65[i] *(1.0)*	1141 *(18.0)*	3137 *(48.0)*
Auxiliary reactor plant equipment [e]	228 *(1.0)*	424 *(1.9)*	1435 *(6.3)*
Heat transport system	89 *(1.0)*	119 *(1.3)*	134 *(1.5)*
BALANCE OF PLANT[f]	517 *(1.0)*	411 *(0.79)*	456[h] *(0.88)*
TOTAL DIRECT COST (M$)	1180 *(1.0)*	2350 *(2.0)*	5948 *(5.0)*
FRACTION OF DIRECT COST DUE TO REACTOR PLANT EQUIPMENT	0.32	0.72	0.79

(a) 1200MW_e. – G. Delene, *Fus. Tech.* 19 807 (1991). (b) 1000MW_e. – *ARIES-I Tokamak Reactor Study* UCLA-PPG-1323 (1991) plus updates via private communications from C.Bathke. (c) ~1000MW_e equiv. (3000MW_{fus}). – L.J. Perkins et al., *ITER EDA Costs*, presentation to US TAC, UCLA (Aug. 1993). (d) Fission :– reactor vessel+internals; Fusion :– FPC out to cryostat. (e) Fission :– control systems, maint. equip., safety systems, fuel handling and storage, I&C; Fusion :– Aux. power, vacuum systems, power supplies, fuel handling, I&C, maint. equip., etc. (f) Electric plant equipment; miscell. plant equipment, heat rejection system, turbine plant equipment (not for ITER). (h) For ITER, also includes hot cell equip., rad waste, fluid supply, magnet fab equip. (g) Includes magnet fab. bldgs. (j) Excludes initial fuel charge; this would be an extra $87M.

The Trouble with Conventional Fusion

The world fusion program has made significant scientific progress over the past decades and is now approaching "break-even" conditions. However, there is a growing realization that our present, conventional approaches may not lead to attractive commercial reactor products able to compete in the energy marketplace of the 21st century for the following reasons:

- Very low power density ($\Rightarrow$ high capital cost per kW produced)
- High complexity ($\Rightarrow$ low perceived reliability/maintainability)
- Pulsed operation for ITER-like reactors
- Minimum unit sizes of $\geq$2GWe
- Not necess. radioactively benign (tritium + neutron-activation for D-T)
- Very high development costs (billions of dollars) for the next stage, notwithstanding that the ultimate power reactors appear uncompetitive

DESIRABLE FEATURES FOR ATTRACTIVE FUSION POWER PLANTS

DESIRED FEATURE	WHY?	HOW?
High Power Density	• Reduce the cost of the fusion power core per kW_e generated, ⇒ lower COE • Economic in smaller unit sizes • Lower cost development path	• High beta; high field • New materials: – high heat-flux – longer lifetime – higher strength • Non-solid first walls • Higher surface/volume configurations • New physics
Simplicity	• Higher availability through increased reliability and maintainability	• Fewer coupled complex systems • No interlinked coils • Minimal external plasma control systems • Steady-state operation • No disruptions • New physics
Steady-State Operation	• No pulsed cyclic fatigue • No thermal storage required	• Efficient current-drive, high bootstrap fraction • No plasma current (stellerator-like)
Low/Zero Radioactive Inventory	• Eliminate short-term decay-heat accident potential • Eliminate long-term waste disposal inventory	• Advanced, low-activation materials • Advanced fuel cycles
Advanced Fusion Fuel Cycles	• No tritium • Low/zero neutron output • Direct energy conversion	• New physics

The Bottom Line :

Key Results

- The single most important physics parameter in which to seek improvements is beta (i.e, β_n/q) in order to increase the power density and, therefore, decrease the cost of the fusion power core. Ultimately, the maximum usable value of β^2 (times B^4) will be limited by the surface heat flux (first wall neutron wall load and/or divertor)

- A surprisingly narrow range of plasma confinement (or beta) is found to be useful in minimizing the *COE* for a tokamak reactor when only modest values in beta (or confinement) are attainable.

- For the plasma beta limited by a Troyon coefficient (β_n) in the range ~4 to 6 (%mT/MA), confinement enhancement *H* factors (ITER89-P scaling) of only ~2 to 2.5 are useable for steady-state driven tokamaks.

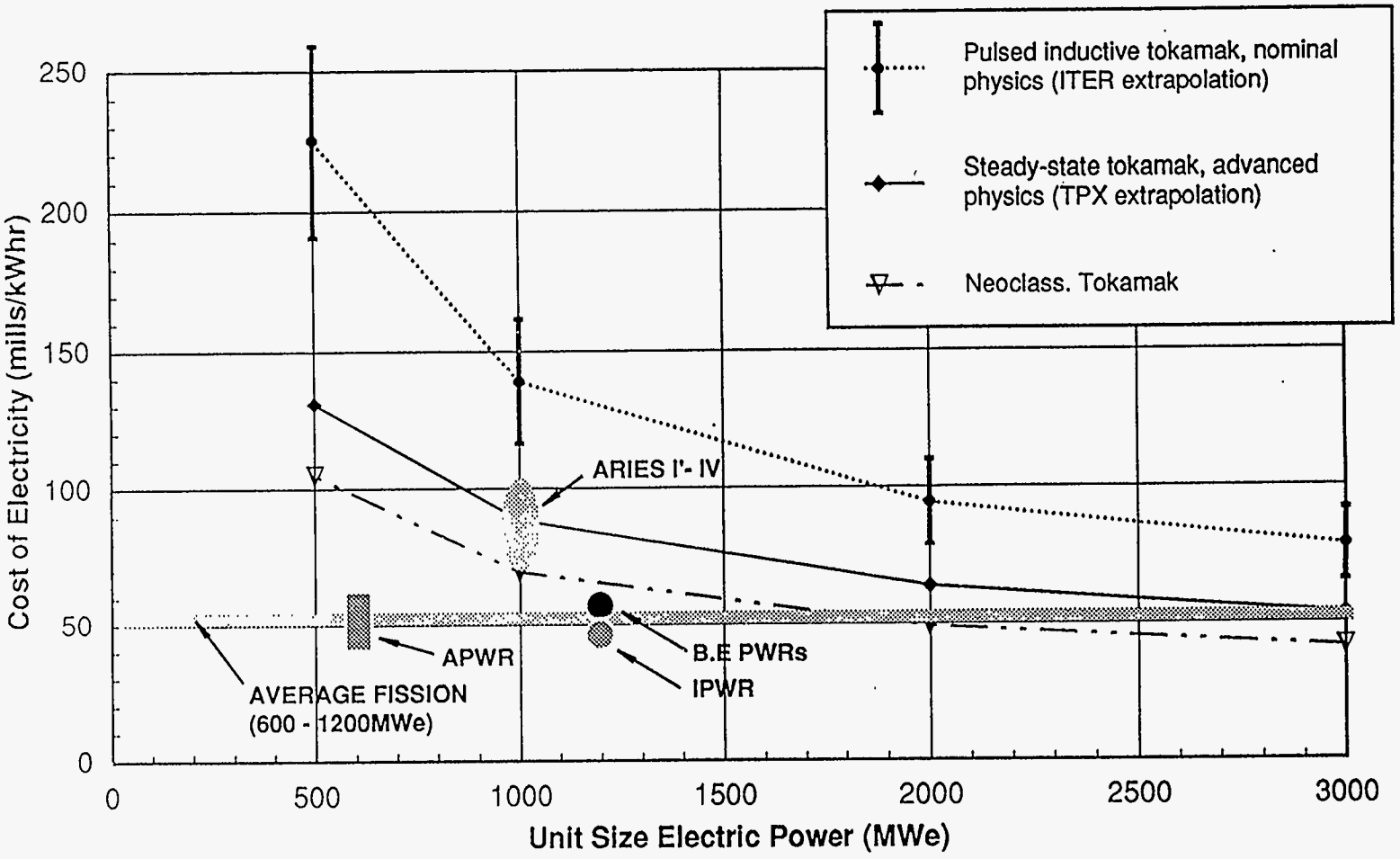
- Increases in current drive efficiency reduce the *COE* by lowering the power recirculating fraction, but do not significantly alter the useful range of confinement and beta.

- Operation at a high edge safety factor q_{95} (e.g., for second stability) results in a minimal increase on the *COE*.

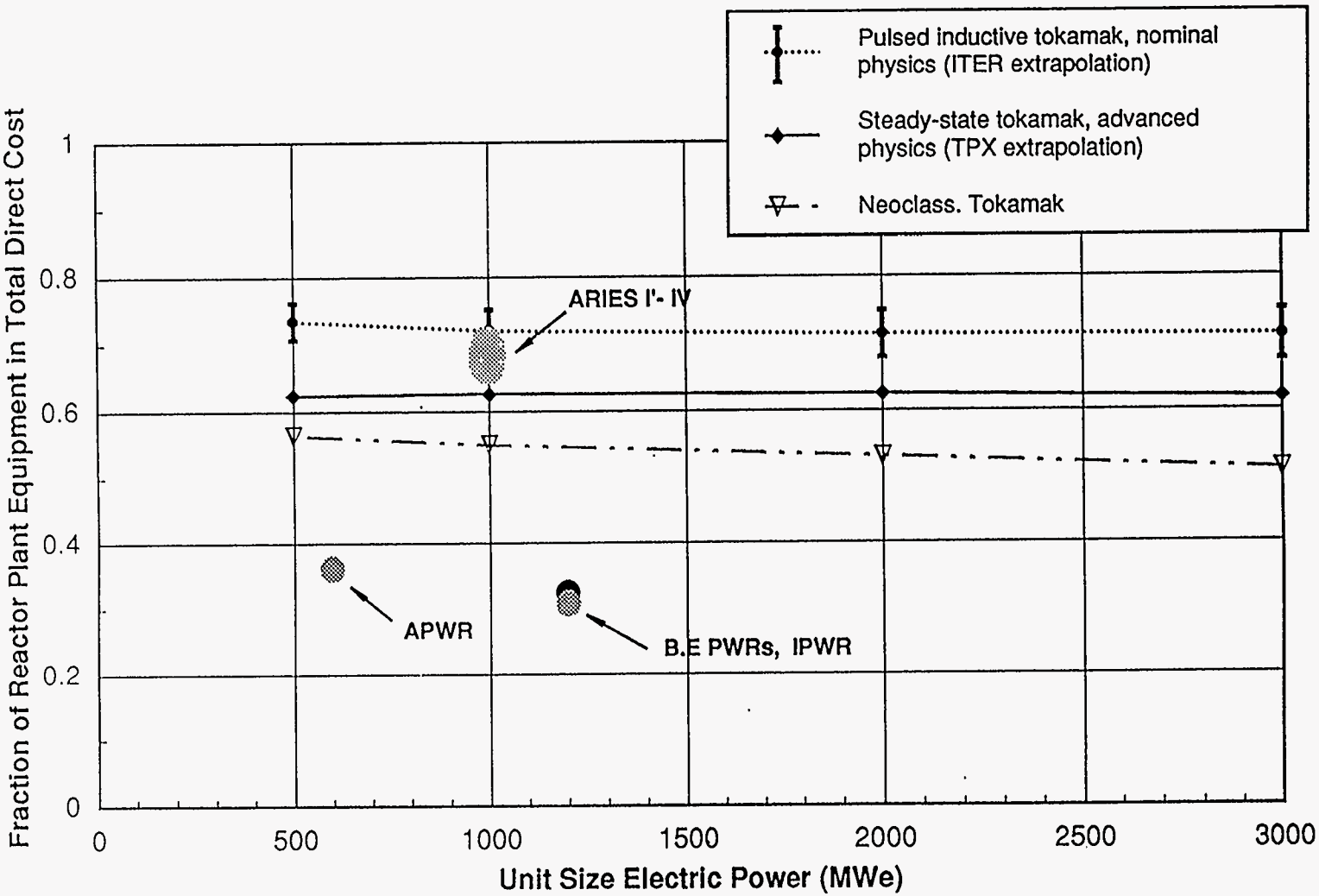
- Inductively driven, pulsed reactors can make use of somewhat increased ranges of confinement, relative to the steady-state cases. For a Troyon beta limit coefficient β_n around 4, *H* factors of up to 2.5 are useable; for β_n around 6, *H* factors up to 3 are useable.

- The economics of an inductively-driven, pulsed reactor is very sensitive to the required pulse length. With comparable *H* factor and β_n levels, pulsed reactors obtain *COE*s comparable to steady-state cases for burn pulse lengths near 5 hours, provided this can be attained without significant penalty for energy storage or pulsed cyclic fatigue (The ARIES/PULSAR Team suggest optimum pulse lengths of ~3 hours -- Sept 93).

- When the usual magnetic confinement constraints (confinement, beta, etc.) are completely removed, small devices are found with similar capital cost, reactor core cost fraction and *COE* as future fission reactors. However, the ability of the solid first wall to withstand the power fluxes (neutrons, radiation, divertor loads) is the key engineering factor in the ability of these "ultimate" reactors to accommodate truly advanced physics

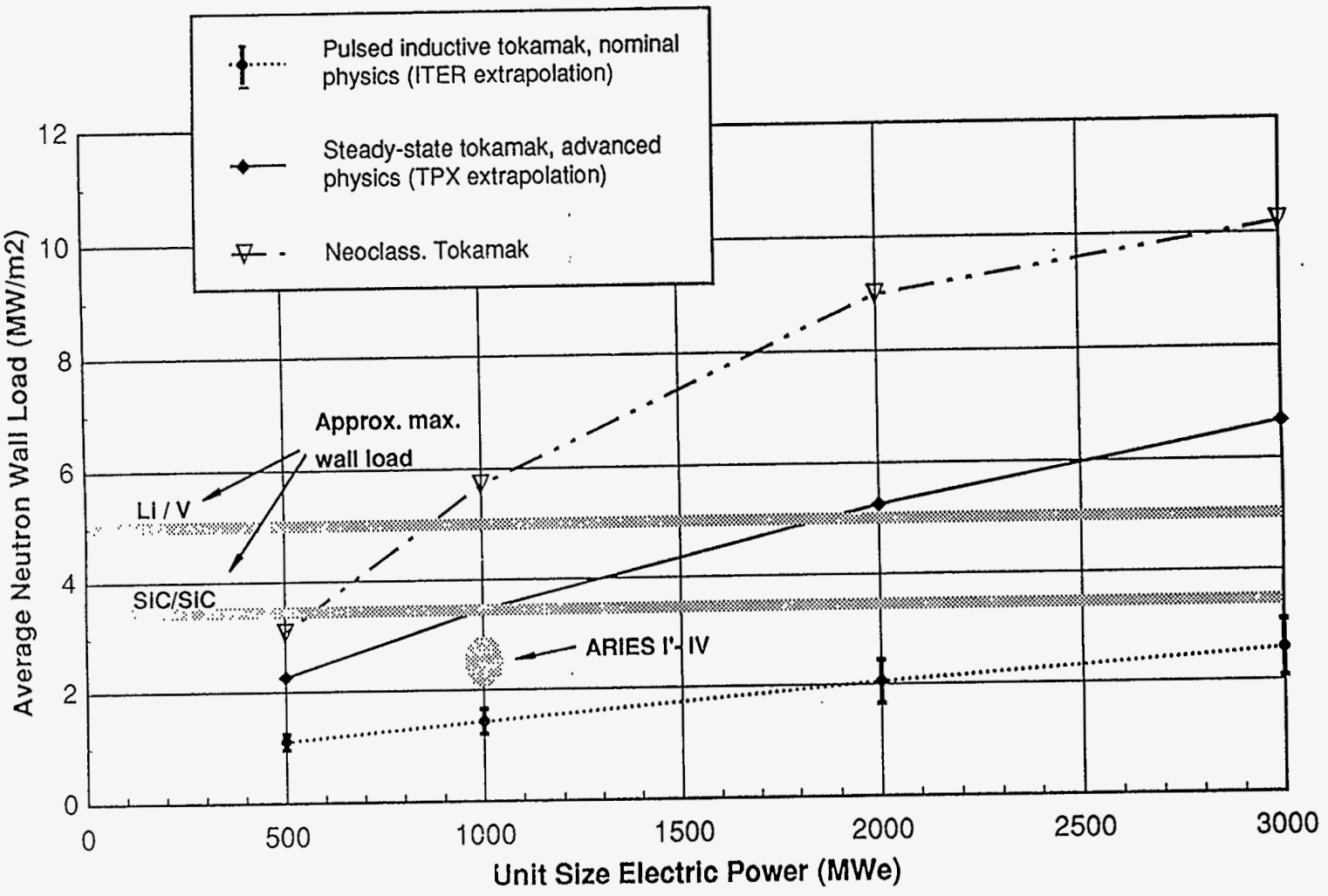


ECONOMY OF SCALE CHARACTERISTICS FOR THE TOKAMAK RELATIVE TO FISSION (1/10th KIND MATURE PLANT, LEVELIZED 1992-$)

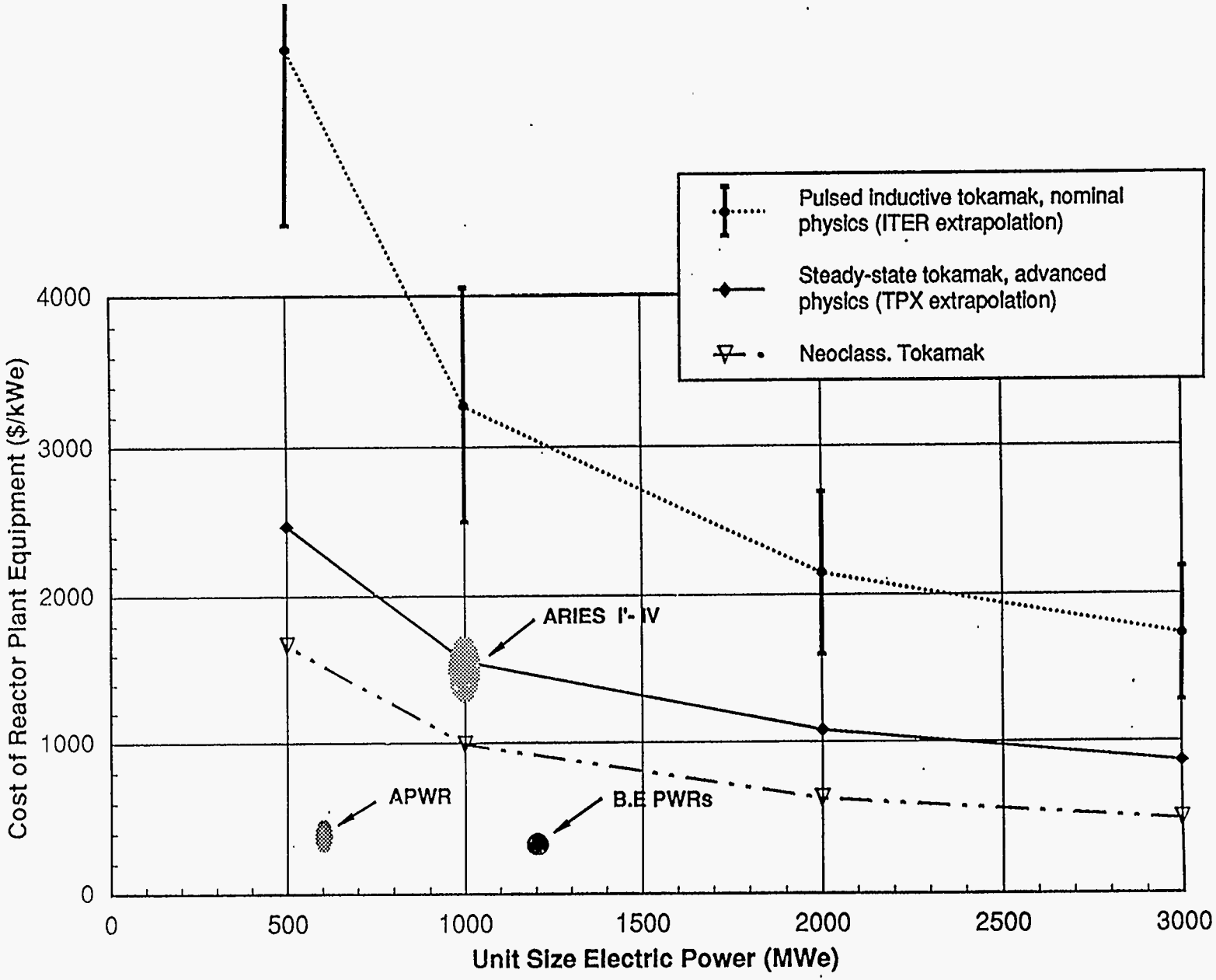


RATIO OF COST OF REACTOR PLANT EQUIPMENT TO TOTAL DIRECT COST FISSION -v FUSION

(RPE = fusion power core/nuclear island + aux. reactor plant equipment + primary heat transport)



RESULTING NEUTRON WALL LOAD -v- NET ELECTRIC OUTPUT



COST OF THE REACTOR PLANT EQUIPMENT IN $/kWe (1992 - $)

(RPE = fusion power core/nuclear island + aux. reactor plant equipment + primary heat transport)

1000MW$_e$ "TOKAMAK" REACTORS: THE IMPACT OF ULTIMATE PHYSICS

	Pulsed Inductive Tokamak[d] Modest phys.[a]	TPX-Extrap Tokamak Advanced phys.[b]	Neoclassical Tokamak Neoclass τ_E, $\beta \leq 1$	Magnetic Toroid No τ_E, $\beta \leq 1$	Ultimate Reactor[c] No physics constraints
Relative COE	1.7 - 2.3	1.3	1.0	0.84	0.70
Mass power density (kW_e/tn)	19 - 26	62	100	210	410
Reactor plant equip. fract.	0.69– 0.75	0.63	0.55	0.43	0.34
R_o (m)	9.0 – 11	5.7	4.2	3.8	1.6[c]
A	3.7 – 4.9	3.5	3.5	6.7	1.0*
I (MA)	18 – 15	11	8.0	N/A	N/A
B (T)	6.3 – 7.3	5.9	4.5	2.8	N/A
q_{95}	3.0* – 3.0*	4.0*	3.0*	N/A	N/A
P_{aux} (MW)	0 – 0	104	0	N/A	N/A
B.S. fract.	0.26 – 0.31	0.71	1.0*	N/A	N/A
Confinement H used	2.0* – 2.0*	2.72	N/A [2.1[f]]	N/A	N/A
Troy. Coef. β_N used	2.5* – 3.0* [d]	6*	N/A [10[f]]	N/A	N/A
β (%)	0.029 – 0.023	0.069	0.15	1.0*	N/A
Neut. wall load (MW/m^2)	1.6 – 1.2	3.5	5.9	12	13
Burn pulse length(hr)	1 – 10[d]	S. State	S. State	S. State	S. State

* -- Parameter at constraint bound or fixed. (a) Modest physics ($H \leq 2$, $\beta_N \leq 2.5$ - 3, 1 – 10hr burn). (b) Advanced physics ($H \leq 4$, $\beta_N \leq 6$, s-state) expected from sucessful TPX program. (c) Can be equally considered point source plasma with FW radius at 1.6 + 0.15(s.o) = 1.75m to keep optimum neut. wall load at $13MW/m^2$. Note this is also the ultimate ICF reactor (∞–rep rate). (d) Cases shown for inductive burn times in the range 1 – 10hr. The ARIES/PULSAR team (Sept 93) suggest optimum pulse length is ~3hr , implying~70,000 pulse fatigue cycles at end-of-life. A 10hr pulse length machine accrues ~20,000 pulse fatigue cycles. (f) H or β_N not constrained here but can be evaluated.

Model :

All Cases Here Were Run with *SUPERCODE* : Our 1-1/2D Optimization Systems Code

- A fast 1-1/2D tokamak physics and engineering simulation code
- Coupled 1-1/2D physics, engineering and costing models.
- Simulation and design optimization for ITER, TPX and commercial reactors
- 1D transport with a choice of models (including RLW ∇T_{crit} transport model)
- 2D variational equilibria
- Simple/detailed (fast/slow) models for IC fast-wave and neutral beam current drive and/or heating.
- Optimization (e.g minimize cost-of-electricity) in multi-variable space (~20-40 independent variables) subject to a user-defined set of inequality constraints (e.g: P_{net}=1000MW_e, $\beta_N \leq 3$, neut. wall load $\leq$ 10 MW/m^2, ..., etc)
- Unique, user-friendly executive shell. Can set up and solve new problem or input your own computational procedure without need to recompile
- Current versions on Sun, HP, IBM RS/6000 and Cray platforms.

FORMULATION OF REACTOR OPTIMIZATION SYSTEM STUDIES WITH *SUPERCODE*

- Decide on the objective function:
  e.g.: Minimize cost of electricity (may be a multi-attribute objective function)

- Decide on the performance requirement:
  e.g.: Net electric output=1000MW$_e$; t_{burn}=10 hr or s.-state current-drive, ..., etc

- Decide on the physics and engineering "rules":
  e.g.: Confinement, beta, q, shielding, TF/OH constraints, etc.
  (These are typically inequalities.)

- Decide on the design parameters we are free to vary; bound them if required,
  e.g.: R, a, A, I, B, n, T, P_{aux} , conf'm'nt H ($\leq H_{max}$), beta β_N ($\leq\beta_{N,max}$)
  radial/vertical builds, TF/OH coil geometry, first wall power fluxes, ..., etc.

⇒ This is the formulation of a constrained, non-linear optimization problem.

⇒ In *SUPERCODE*, all the performance requirements and physics/engineering rules form a global set of inequality constraints, and are solved silmultaneously.

⇒ All design parameters form a global variable set and are iterated simultaeously; they can be bounded if required.

⇒ The detailed physics and engineering models in *SUPERCODE* are effectively function evaluators for the constraint set.

Other Constraints and Assumptions

- Major physics assumptions as above, otherwise use standard TPX/ITER physics "rules" and models.
- 0-D physics and 1-D current drive (NB).
- Use ITER cost database for direct costs ($\Rightarrow$ will obtain first-of-a-kind COE numbers).
- Use ARIES/GENEROMAK-like cost database and models for recirculating powers, indirect costs, interest, construction time and operating costs, etc (see below).
- All engineering constraints and models as ITER [Nb_3Sn magnet allowables (stress, protection, stability, . . .), shielding, builds, etc.].
- Shielding determined by 30-yr fluence lifetime of superconducting magnets.
- Blanket lifetime, changeout time and yearly costs (fluence limit = 20 $MWyr/m^2$) depend on neutron wall load.
- Neutron wall loading ≤ 20 MW/m^2 (never reached)
- No divertor constraints.

Constraints

Physics
plasma power balance (with ITER89-P confinement)
Troyon beta limit
$q95 \geq 3$
$q0 = 1.05$
$I_{plasma} = I_{bootstrap} + I_{noninductive}$ (a)
$(V\text{-sec})_{capability} \geq (V\text{-sec})_{requirement}$ (b)
$I_{plasma} \geq$ minimum to confine alphas

Magnets
TFC conduit stress $\leq$ 550 MPa
TFC external case stress $\leq$ 550 MPa
TFC dump voltage $\leq$ 20 kV
TFC $I_{operate} / I_{critical} \leq 0.5$
TFC hot spot temp $\leq$ 150 K
TFC temperature margin $\geq$ 2 K
OHC $J_{BOP} \leq$ allowable
OHC $J_{EOF} \leq$ allowable

Device
inner shield thickness $\geq$ allowable
inner divertor to TFC distance $\geq$ allowable
TFC ripple $\leq$ allowable
Av. neutron wall load $\leq$ 20 MW/m^2 (never reached)
area for injection power < 10% wall area
$P_{net\text{-}electric}$ = input requirement

a - This constraint is only applied for steady-state cases.
b - The Volt-second requirement is different for the pulsed and steady state reactors.

Fixed Assumptions

Plasma	
plasma shape	Double Null
elongation (x-point)	2.0
triangularity (x-point)[a]	$0.3 \leq \delta \leq 0.8$
density profile exponent α_n	0.5
temperature profile exponent α_T	1.0
Radial Builds (m) :	
OHC / TF Coil gap	0.08
TF coil / VV gap	0.07
inner vacuum vessel	0.32
inner breeding blanket zone	0.19
first wall	0.04
inner/outer scrapeoff	0.15
outer breeding blanket zone	0.25
outer shield	0.90
outer vacuum vessel	0.32
Outer VV/TFC gap (b)	$\geq$ 0.25
Vertical Builds (m) :	
Shield on Top (c)	0.75
divertor structure	0.35
gap	0.25
Divertor X-point to Strike-point - outer / inner (m)	1.5 / 0.7

Optimization Variables

Physics
plasma temperature
plasma density
plasma current
current drive power (MW)
heating only power (MW)
triangularity

Overall Device
major radius
minor radius
field on axis
inner shield thickness
inner scrapeoff (m)
outer TF/shield gap (m)

TF Coil
overall current density
conduit thickness (mm)
external case thickness
current per turn (kA)
conductor Cu fraction
dump time (s)
TF coil total thickness

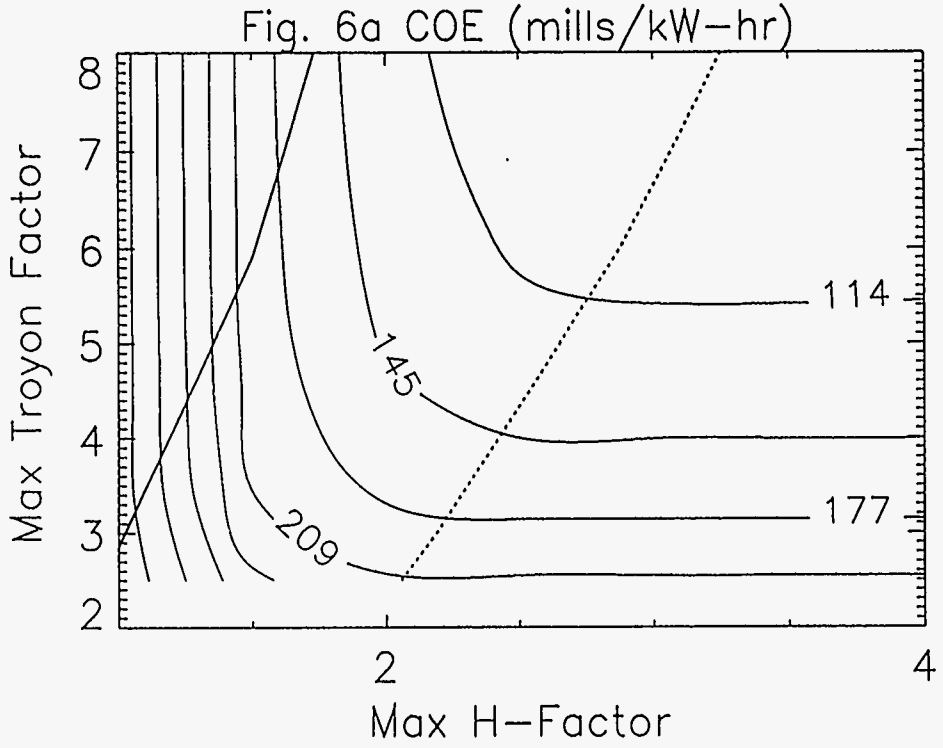
OH Coil
current density @ BOP
current density @ EOF
transformer hole(m)
winding thickness (m)

Assumptions for Power Balance, and Cost of Electricity Calculation

	This Study	Generomak	ARIES-1
Plant Power Balance			
thermal to electric efficiency	0.4	0.36	0.49
non-neutron power to thermal power	70%	70%	100%
blanket energy gain	1.30	1.14	1.30
injector power efficiency, wall plug to plasma efficiency	72%	100%	71%
Costing assumptions			
Construction time (y)	6	8	6
plant life (y)	30	30	30
average capacity factor	75%	65%	76%
indirect + contingency cost factor	55%	50%	45%
fixed charge rate($FCR0$)	0.0966	0.10	0.0966
effective cost of money (y^{-1})	0.0605	0.09	0.0435
inflation rate (y^{-1})	0.05	0.05	0.06
direct cost methodology	1st of-a-kind and 10th of-a-kind	10th of-a-kind	10th of-a-kind

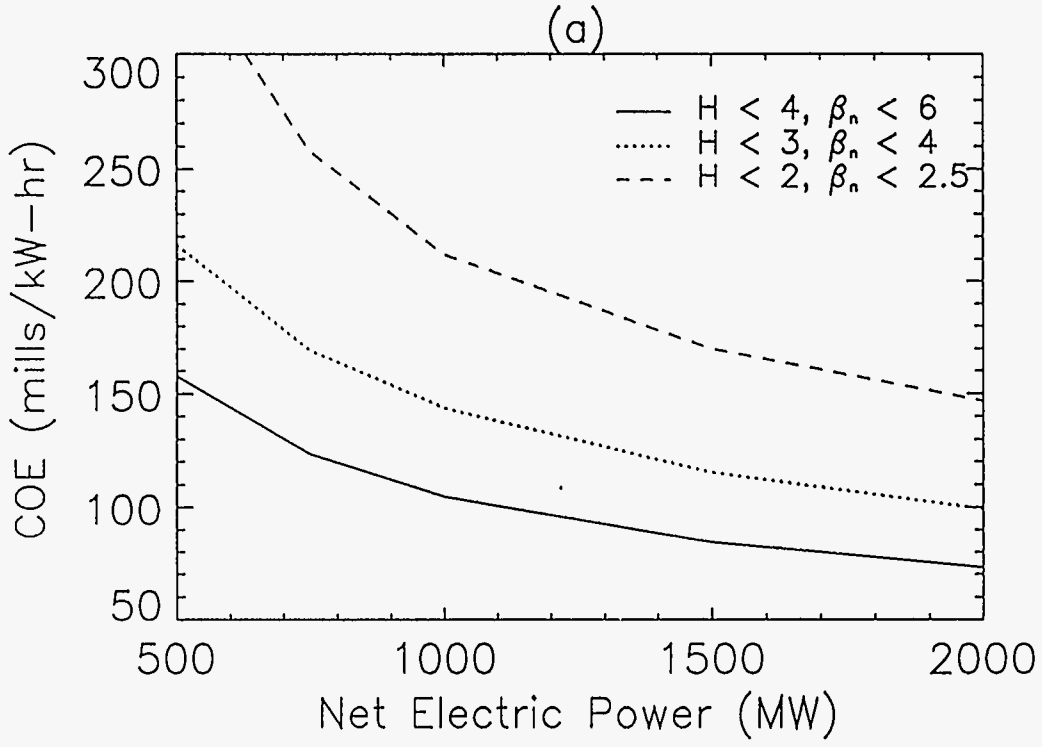
Pulsed Inductive Reactors:

Pulsed Inductive Reactor (H,β_n) Space - Minimum COE



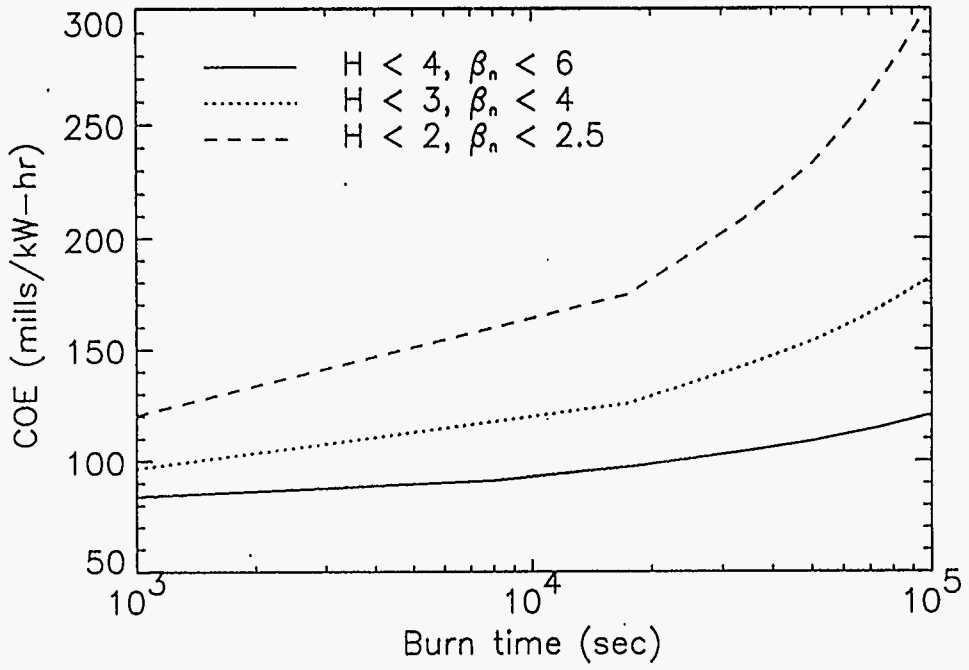
- There is a narrow region of useful (H,β_n) space.
- Higher H factors are useful for a given beta limit, relative to the steady-state cases .

Pulsed Inductive Reactor: Economy of Scale -v- $P_{net,e}$



- Pulsed reactors show a stronger economy of scale than the steady-state reactors.

Pulsed Inductive Reactors: Impact of Pulse Length



- There is a critical burn time, above which the COE increases rapidly. For This is around ~ 10^4 secs (~3 hours).

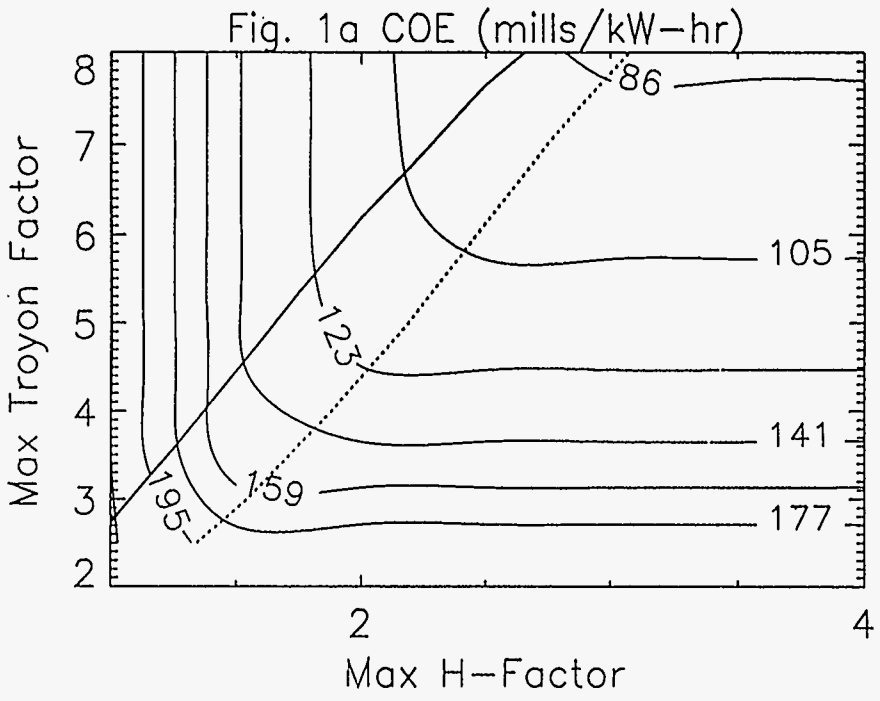
Pulsed Inductive Reactors: Min. COE Reactor Parameters.

	1000 MW_e				500 MW_e		2000 MW_e	
Pulse Length:	10 hrs	10 hrs	5 hrs	5 hrs	10 hrs		10 hrs	
Physics Limits:	$\beta_n \leq 4$ $H \leq 3$	$\beta_n \leq 6$ $H \leq 4$	$\beta_n \leq 4$ $H \leq 3$	$\beta_n \leq 6$ $H \leq 4$	$\beta_n \leq 4$ $H \leq 3$	$\beta_n \leq 6$ $H \leq 4$	$\beta_n \leq 4$ $H \leq 3$	$\beta_n \leq 6$ $H \leq 4$
COE (mills/kW hr)	144	105	126	97.5	216	158	99.3	73.0
Capital Cost (B$)	8.44	5.78	7.16	5.24	6.48	4.49	11.3	7.73
$f_{recirculate}$ (%)	12.6	12.6	12.6	12.6	15.9	15.7	10.5	10.5
Core Mass (kTonnes)	36.3	23.6	32.8	20.0	30.4	17.6	56.1	31.6
MPD (kW_e/tonne)	27.5	42.4	30.5	50.0	15.2	28.4	35.7	63.3
Device Core Cost (%)	75.7	68.2	72.3	65.6	75.7	69.0	74.3	66.2
Major radius (m)	8.91	7.18	8.14	6.70	8.13	6.57	9.99	8.00
Aspect ratio	5.41	5.31	4.42	4.65	5.50	5.50	4.63	4.85
Plasma Current (MA)	9.96	7.84	12.2	8.85	7.99	6.19	15.1	10.9
Field on axis (T)	7.48	6.92	6.45	6.25	6.97	6.56	6.94	6.79
B max-TF coil (T)	11.3	11.2	10.7	10.9	10.7	10.8	10.8	11.0
q95	3.0 *	3.0 *	3.0 *	3.0 *	3.0 *	3.0 *	3.0 *	3.0 *
Injection power (MW)	0	0	0	0	0	0	0	0
Bootstrap fraction	0.509	0.754	0.501	0.718	0.526	0.783	0.471	0.714
Confinement *H* used	2.42	2.86	2.40	2.87	3.0 *	3.57	1.97	2.32
Troyon Coef. β_n used	4.0 *	6.0 *	4.0 *	6.0 *	4.0 *	6.0 *	4.0 *	6.0 *
$\langle T_e \rangle_n$ (keV)	16.3	15.9	16.5	15.4	16.1	15.8	18.1	16.7
$\langle n_e \rangle$ (10^{20} m^{-3})	1.35	1.84	1.26	1.83	1.14	1.57	1.29	1.94
Beta (%)	3.23	5.04	3.46	5.90	3.10	4.73	4.03	5.83
Wall load (MW/m^2)	2.02	3.03	1.99	3.04	1.27	1.90	2.72	4.38

* -- Parameter at constraint bound or fixed

Steady-State Reactors:

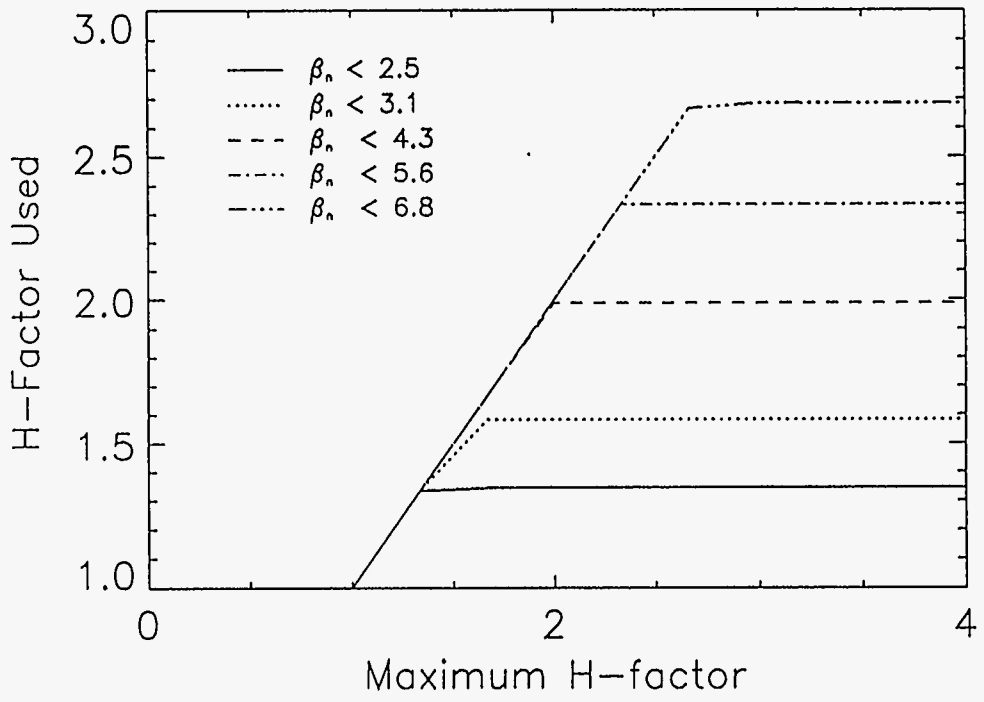
Steady-State Reactor (H,β_n) Space - Minimum COE



- There is only a narrow region of useful (H, β_n) space.
- Improvements over present-day beta of β_n ~ 2.5-3 offer more benefit than improvements over present-day confinement of H ~ 2

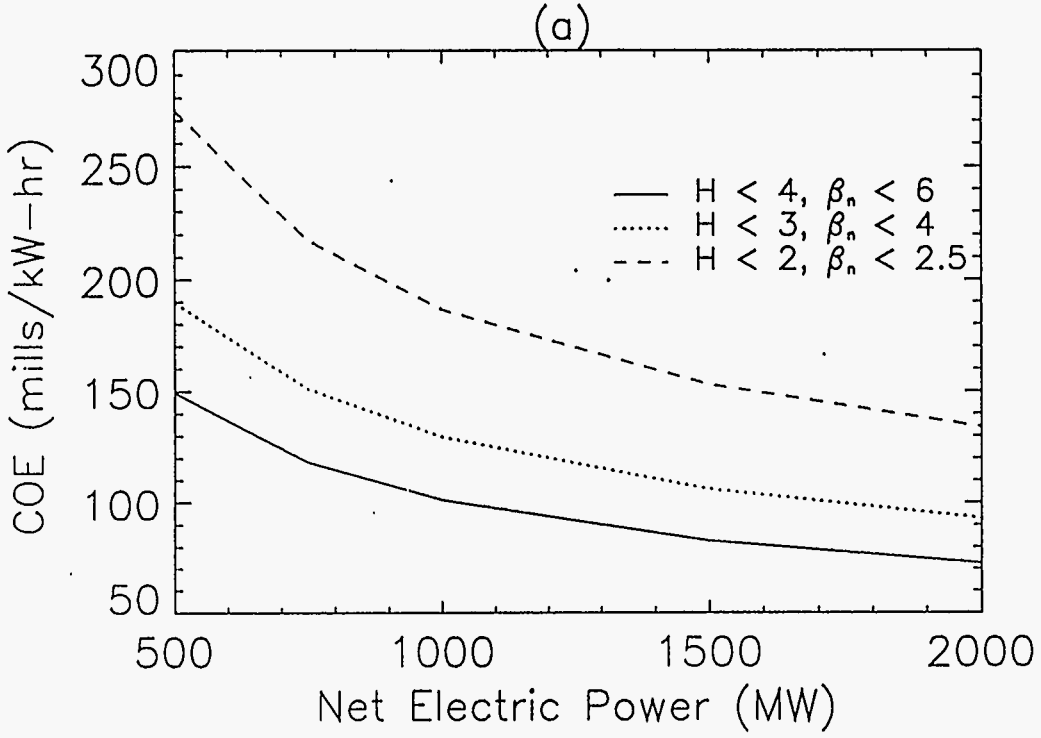
Steady-State Reactors: Useful H-factor range

(at different Troyon beta levels)



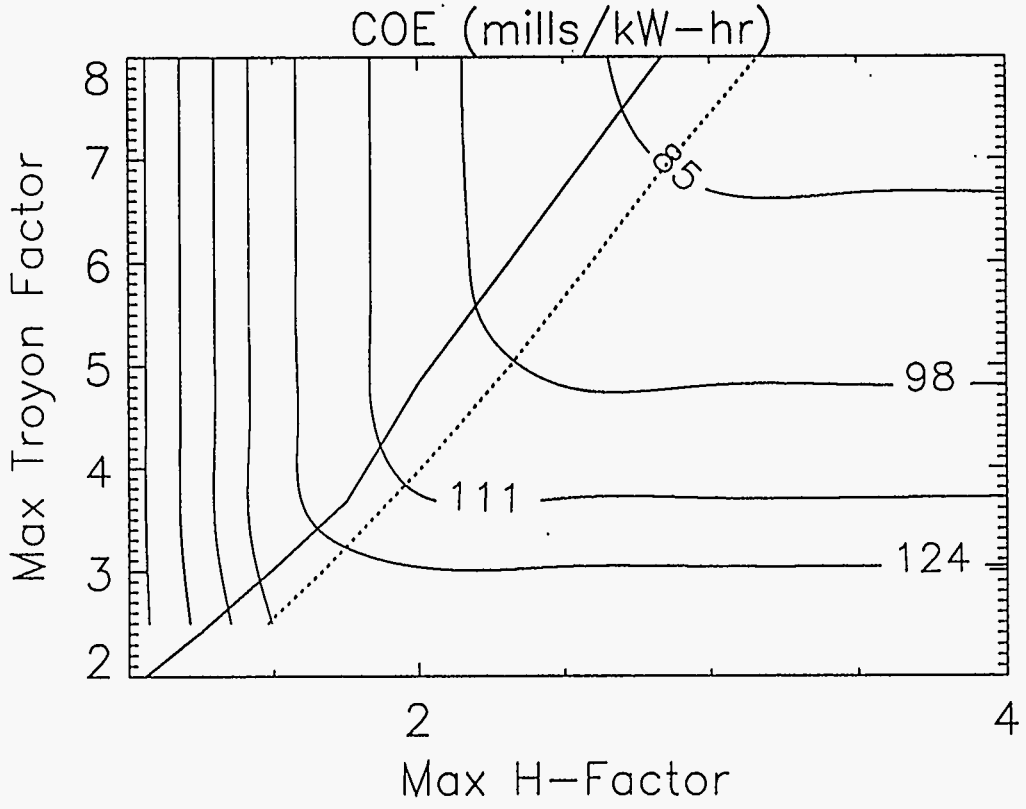
- For $\beta_n \leq$ 3-4 , confinement enhancement factors (ITER89-P) of only H ~ 1.5 to 2 are needed.

Steady-State Reactor: Economy of Scale -v- $P_{net,e}$



- Need advanced physics (i.e. β_n up to 6, H up to 3) for COE < 100 mills/kW-hr (< 80 mills/kW-hr 10th-kind)

Impact of Current Drive Efficiency
(3x nominal efficiency)



- COE is reduced, but the useful (H,β_n) space is similar to the nominal case.

Steady-State Current Driven Reactors: Min. COE Reactor Parameters.

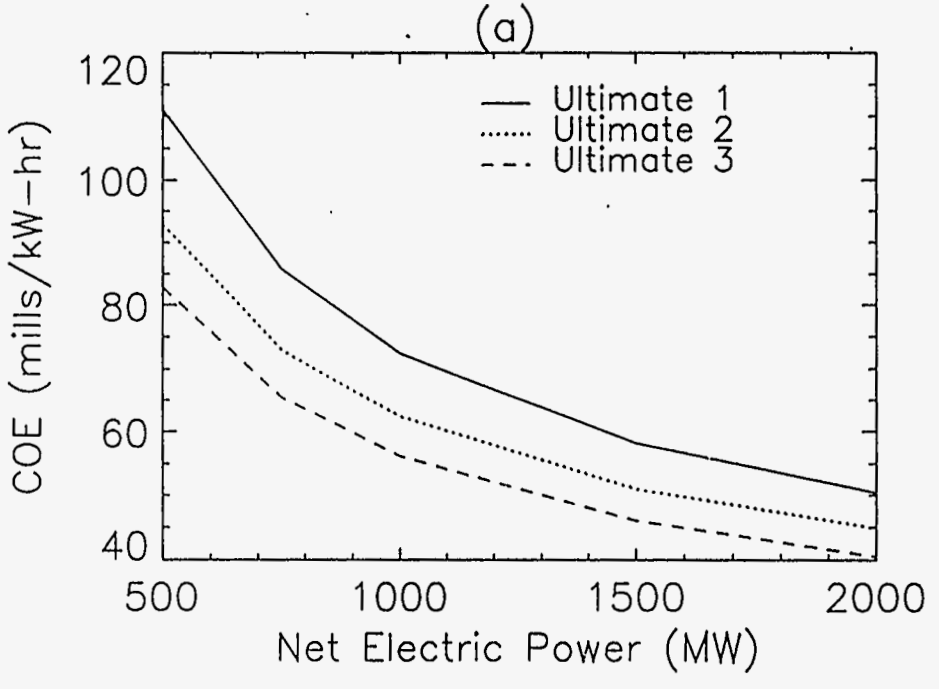
	1000 MW_e				500 MW_e		2000 MW_e	
	$\beta_n \leq 4$ $H \leq 3$ $q_{95} \geq 3$	$\beta_n \leq 6$ $H \leq 4$ $q_{95} \geq 3$	$\beta_n \leq 4$ $H \leq 3$ $q_{95} \geq 5$	$\beta_n \leq 6$ $H \leq 4$ $q_{95} \geq 5$	$\beta_n \leq 4$ $H \leq 3$ $q_{95} \geq 3$	$\beta_n \leq 6$ $H \leq 4$ $q_{95} \geq 3$	$\beta_n \leq 4$ $H \leq 3$ $q_{95} \geq 3$	$\beta_n \leq 6$ $H \leq 4$ $q_{95} \geq 3$
COE (mills/kW hr)	130	102	142	106	190	149	93.0	72.5
Capital Cost (B$) (a)	7.24	5.38	8.09	5.68	5.44	4.12	10.1	7.42
$f_{recirculate}$ (%)	33.2	24.6	31.4	21.4	38.9	29.1	28.4	20.9
Core Mass (kTonnes)	24.5	16.0	31.9	20.1	18.8	12.5	32.4	21.5
MPD (kW_e/tonne)	40.8	62.5	31.3	49.8	26.6	40.0	61.7	93.0
Device Core Cost (%)	71.8	65.6	73.2	66.9	71.8	66.0	71.5	64.4
Major radius (m)	6.27	5.59	6.84	5.91	5.71	5.09	6.98	6.20
Aspect ratio	3.14	3.59	2.87	3.30	3.25	3.73	3.07	3.50
Plasma Current (MA)	19.6	12.3	16.3	10.2	15.4	9.54	24.7	15.8
Field on axis (T)	6.17	5.86	6.16	6.04	5.96	5.60	6.43	6.17
$B_{max\text{-}TF\ coil}$ (T)	13.5	12.5	13.7	13.2	13.2	12.2	13.7	12.8
q_{95}	3.0 *	3.0*	5.0 *	5.0 *	3.0 *	3.0 *	3.0 *	3.0 *
Injection power (MW)	233	122	206	84.5	147	74.9	373	201
Bootstrap fraction	0.393	0.647	0.481	0.767	0.409	0.672	0.381	0.624
Energy Gain, *Q*	12.7	21.9	14.0	30.5	10.9	19.0	14.9	25.5
Conf.inement *H* used	1.90	2.47	2.19	2.96	2.26	2.97	1.58	2.03
Troyon Coef. β_n used	4.0 *	6.0 *	4.0 *	6.0 *	4.0 *	6.0 *	4.0 *	6.0 *
$\langle T_e \rangle_n$ (keV)	22.4	19.0	17.8	15.2	22.0	18.1	22.7	19.8
$\langle n_e \rangle$ (10^{20} m^{-3})	1.27	1.74	1.14	1.67	1.12	1.55	1.44	1.98
Beta (%)	6.36	8.11	4.43	5.65	5.89	7.49	6.77	8.69
Wall load (MW/m^2)	3.00	3.84	2.26	3.06	2.02	2.53	4.45	5.86

* -- Parameter at constraint bound or fixed

"Ultimate" Reactors:

"Ultimate Reactors"

Ultimate 1 - Neoclassical tokamak: No H /β_n limits -- Only Neoclassical confinement and $\beta \leq 1$
Ultimate 2 - Magnetic Toroid: No confinement requirement ($\Rightarrow$ no plasma current), $\beta \leq 1$
Ultimate 3 - Toroidal/point source reactor: No β limit ($\Rightarrow$ no magnets)



- As the confinement and beta limits are completely removed, COEs approaching fission reactors are approached.

"Ultimate" Reactors: Min. COE Reactor Parameters.

	Nominal Steady State	*Ultimate 1 Neoclassical Tokamak*		*Ultimate 2 Magnetic Toroid*		*Ultimate 3 Toroid/Point Source*	
Power Level	*1000 MW_e*	*500 MW_e*	*1000 MW_e*	*500 MW_e*	*1000 MW_e*	*500 MW_e*	*1000 MW_e*
COE (mills/kW hr)	130	111	72.4	92.9	62.3	82.8	56.1
Capital Cost (B$)	7.24	2.85	3.44	2.20	2.72	1.84	2.32
$f_{recirculate}$ (%)	33.2	15.7	12.6	15.7	12.6	15.7	12.6
Core Mass (kTonnes)	24.5	7.31	8.5	3.09	3.73	1.94	2.63
MPD (kW_e/tonne)	40.8	68.4	118	162	268	258	381
Device Core Cost (%)	71.8	57.3	53.0	54.2	43.4	41.8	36.6
Major radius (m)	6.27	4.03	4.23	3.35	3.75	1.29	1.59
Aspect ratio	3.14	3.49	3.60	6.81	6.65	1	1
Plasma Current (MA)	19.6	6.96	7.95	---	---	---	---
Field on axis (T)	6.17	4.03	4.53	2.53	2.81	---	---
B max-TF Coil (T)	13.5	11.1	11.9	6.10	6.23	---	---
q95	3.0 *	3.0 *	3.0 *	---	---	---	---
Inj. power (MW)	233	0	0	0	0	0	0
B.S. fraction	0.393	1.00	1.00	---	---	---	---
$\langle T_e \rangle_n$ (keV)	22.4	7.93	7.59	12.6	15.1	---	---
$\langle n_e \rangle$ (10^{20} m^{-3})	1.27	3.87	5.43	6.35	6.37	---	---
Beta (%)	6.36	14.4	15.3	100 *	100 *	---	---
Wall load (MW/m^2)	3.00	3.20	5.81	7.58	11.7	9.94	12.6

* -- Parameter at constraint bound or fixed

Our Thesis

- Any breakthrough for economic fusion will lie in the investigation of new, alternative physics approaches rather than in refined engineering concepts for the present conventional methods.
- This is particularly true if we are ever to realize economic viability with the advanced (non-D-T) fusion fuels.
- A factor of ~2–10 is required in power-density and reduction of complexity, and not the 10's of % envisioned for the conventional approaches.
- The most fruitful areas for advanced fusion research may include:
  - A revisiting of some older ideas which indicate good reactor potential but which were cancelled prematurely (e.g, the spheromak)
  - Novel, medium-high density, magneto-inertial schemes
  - Novel, non-thermonuclear schemes

"It's no use saying, 'We are doing our best'. You have got to succeed in doing what is necessary"

WINSTON S. CHURCHILL

"The reasonable man adapts himself to the world; the unreasonable man persists in trying to adapt the world to himself. Therefore, all progress depends on the unreasonable".

GEORGE BERNARD SHAW

Our Recommendations

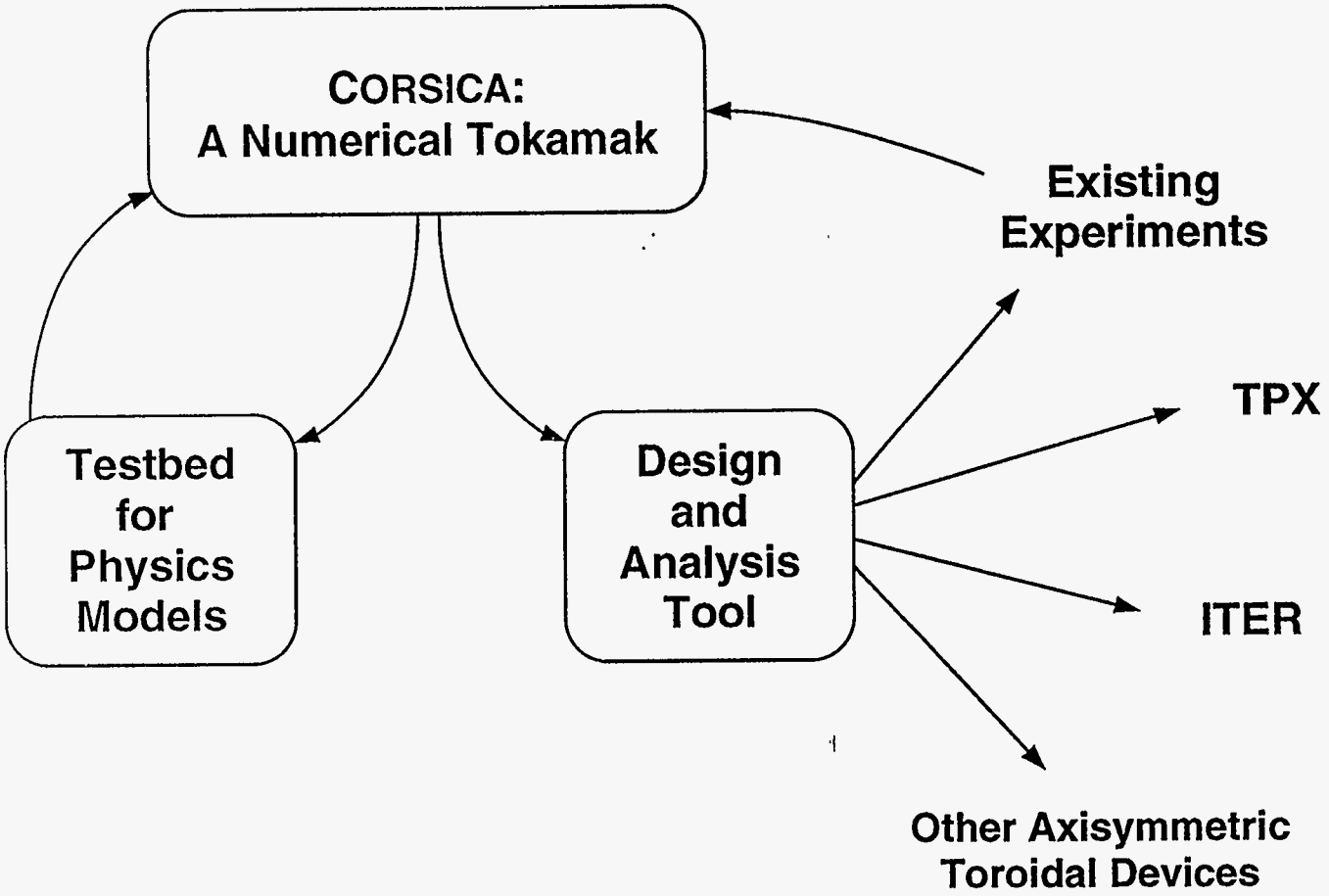
- Acknowledge the need to establish continuing research into new and substantially different fusion concepts. We're not satisfied with our present, conventional approaches!

- Earmark funds for conceptual development and computational modeling, to the stage where new, definitive experiments can be defined.

- Promote intellectual stimulation in breadth, by encouraging high risk, high payoff approaches with the acknowledgment that only a very few may ultimately be successful.

- Couple the prospective advanced physics with an engineering realization to clearly identify the potential for a step change in capital costs, complexity and development path relative to our present concepts. (e.g, Assume the advanced physics works -- how does it make a better reactor?)

CORSICA:
A tool for design, theory, and experimental analysis.



Motivation.

- Future tokamaks more difficult to design:
  - ⇨ Complicated geometry,
  - ⇨ "Advanced tokamak" profiles (inverted q, high bootstrap, etc.),
  - ⇨ Divertor design is extremely important,
  - ⇨ Turbulence modeling important but difficult - need proven analytical models developed from simulation and experiment.

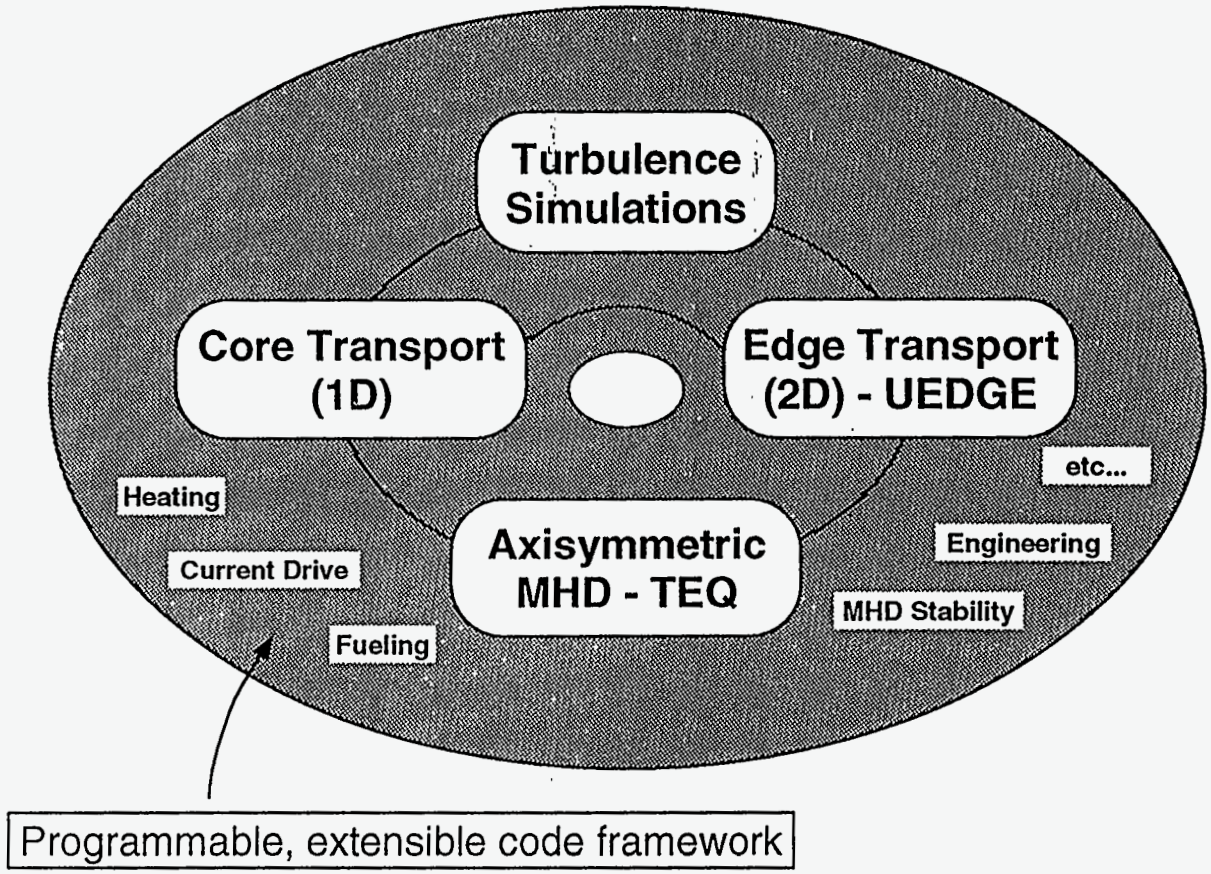
- Machines are expensive and money is tight.

- Need a "fast," flexible, and accurate tool that can be used for comprehensive simulation of the tokamak designs.

- Need to have a test-bed for physics modules where they can be tested in a self-consistent simulation code.

☞ **Need a Numerical Tokamak**

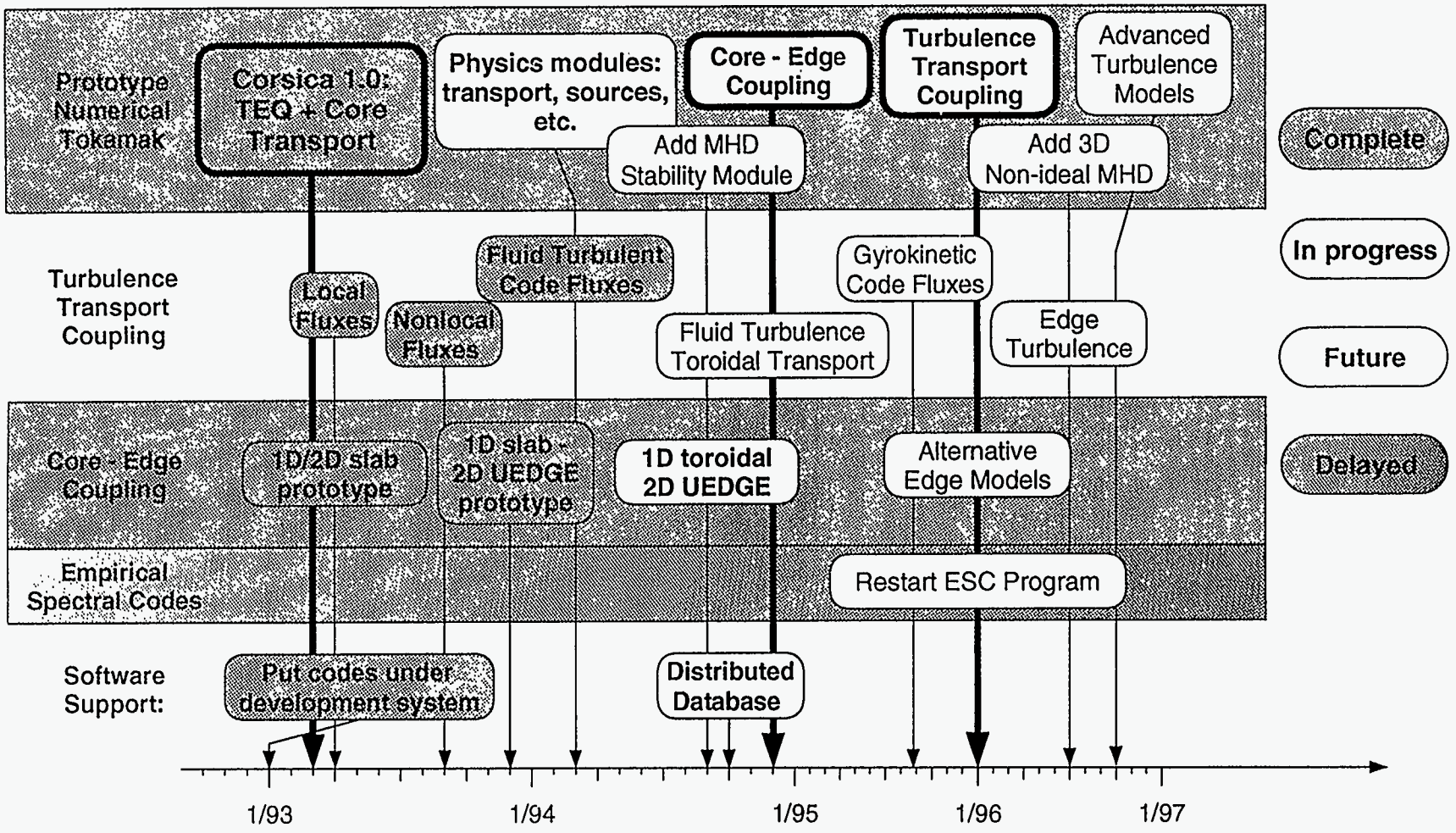
CORSICA's blueprint for a numerical tokamak.



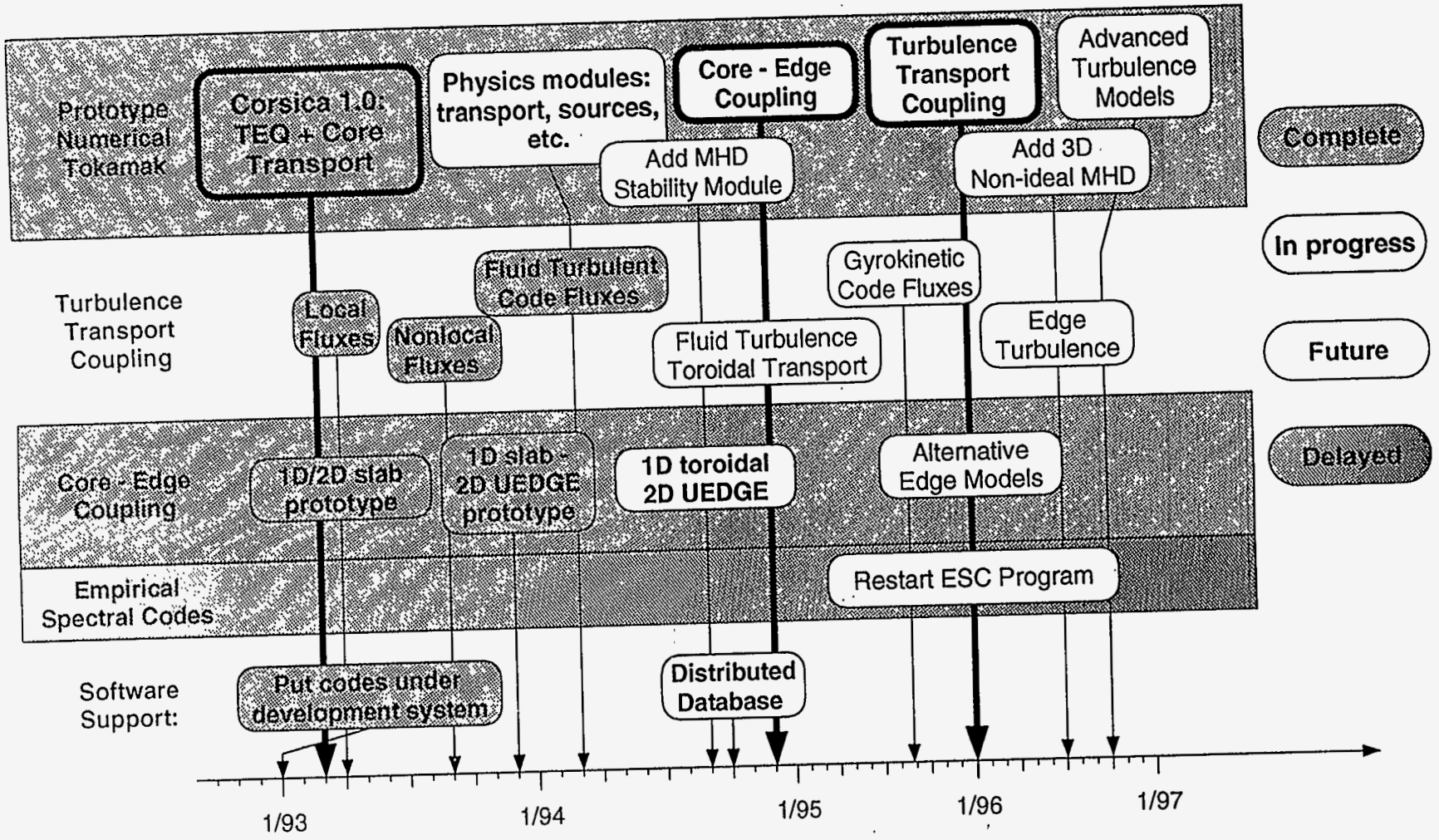
Exploiting timescale disparities.

- For a "numerical tokamak" to be useful, it must be capable of simulating a tokamak on the energy confinement timescale.
- Many processes, such as turbulence and edge transport, operate on *much* shorter timescales.
- These timescale disparities must be exploited.
- The core transport algorithm must be implicit. (Numerical stability at large timesteps requires that various terms be evaluated at the advanced timestep.)

Timeline for milestones



Timeline for milestones



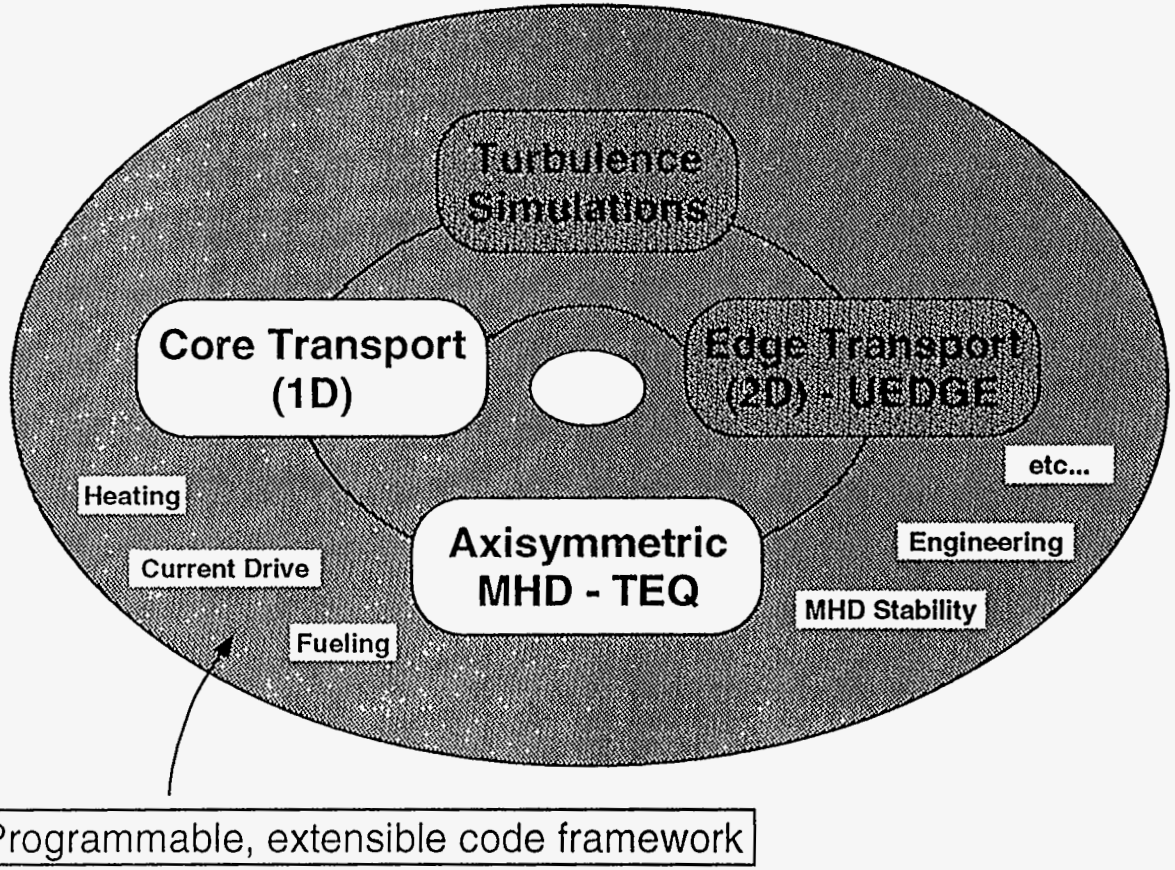
CORSICA:
Contributions and collaborations.

- Current drive - T. K. Mau (UCLA), possibly Ken Kupfer (GA)
- Neutral beams - NFREYA (PPPL)
- MHD, rotation - Alan Turnball (GA)
- MHD inverse equilibrium - from DINA code (Russia)
- Edge code - UEDGE
- Benchmarking with PRETOR - Dominique Boucher (ITER)
- Benchmarking with DINA - Rustam Khayrutdinov (Kurchatov)
- Control - Dave Humphreys (GA)
- Turbulence simulation - ???
- TEQ
- Pieces of core transport from ITER/TPX systems code

Programmable, extensible code framework.

- Programmable:
  - ⇨ Much of the core transport is, in fact, written in a Fortran-like script language that is executed by an interpreter at runtime.
- Modular:
  - ⇨ Organized into "physics packages."
- Extensible:
  - ⇨ Adding new "packages" is relatively easy.
- Interactive:
  - ⇨ Has a command-line interface that can access variables and functions in the compiled packages.
  - ⇨ Can enter statements and functions written in the script language.
  - ⇨ The "main program" is replaced by either a script or by the user interacting directly with the interpreter (à la Mathematica), giving the user great flexibility in how the code is run.
  - ⇨ Also has interactive graphics.
- Portable:
  - ⇨ Runs on Crays and most UNIX platforms.
  - ⇨ Output data format is also portable.

CORSICA 1: Coupling transport to MHD equilibrium.



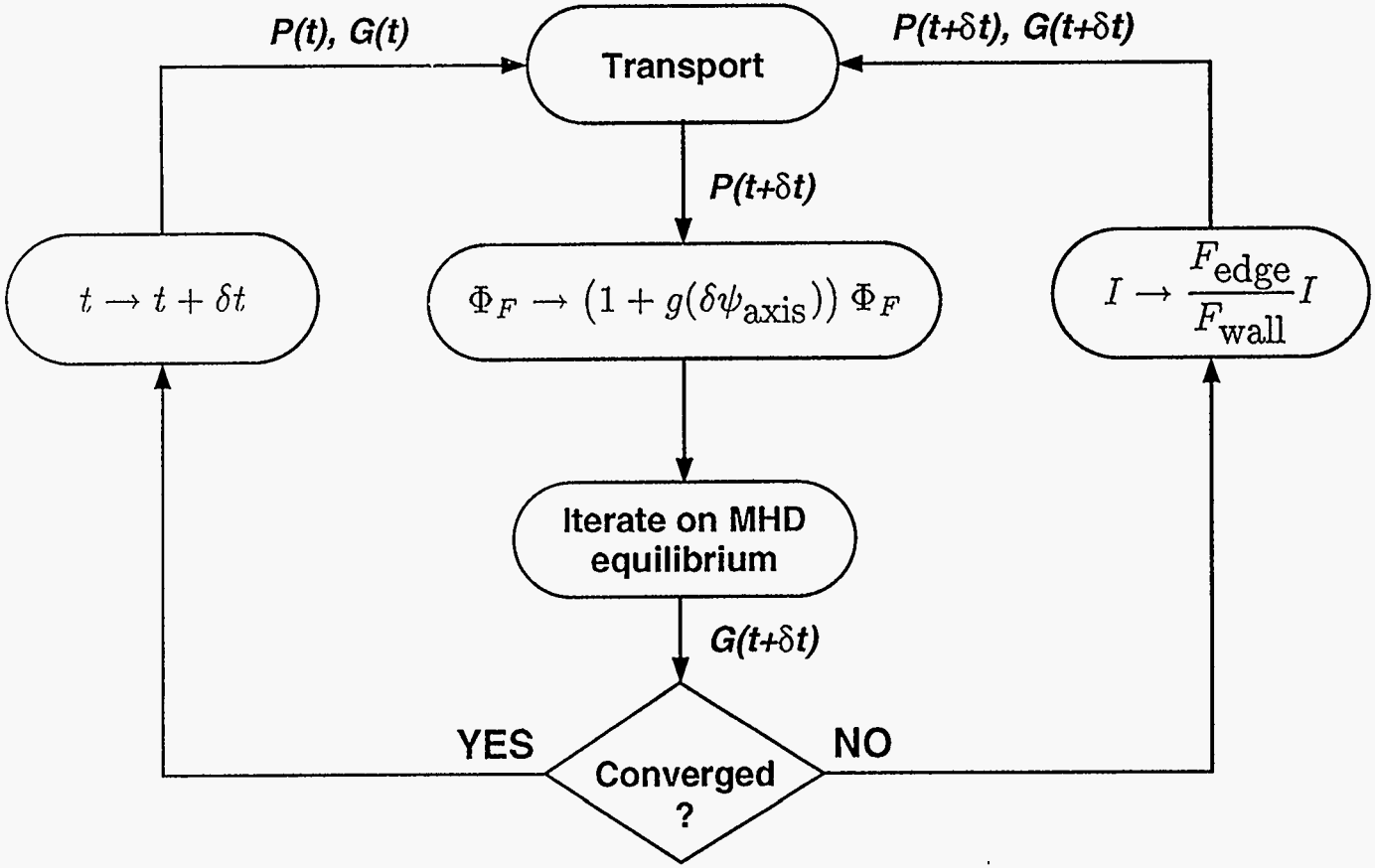
Coupling transport to MHD equilibrium.

- Fast time-scales are avoided:
  - ⇨ quasi-static MHD equilibria (avoid Alfvèn timescale!)
  - ⇨ 1-D transport (avoid parallel transport timescale).

- Converge transport and equilibrium iterations simultaneously (inefficient to converge equilibrium every transport iteration).

CORSICA 1:
Coupling transport to MHD equilibrium.

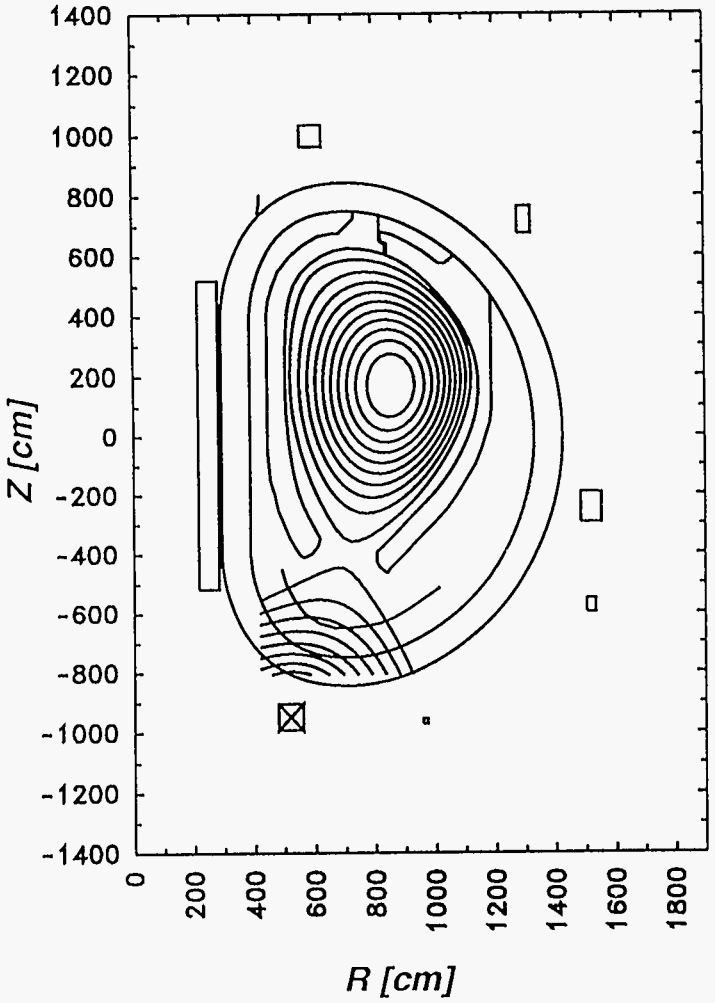
P = Profiles
G = Geometry

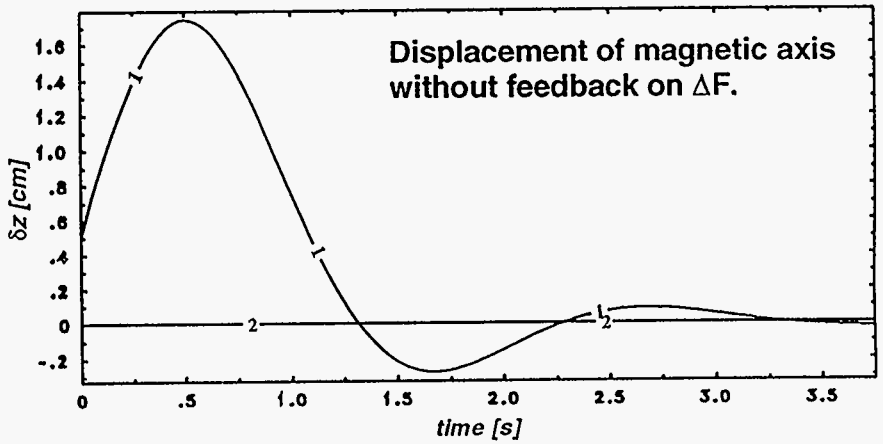


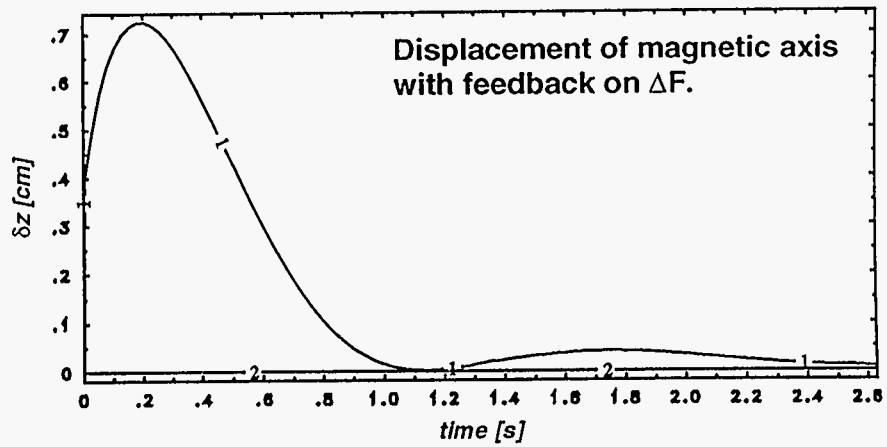
CORSICA 1 (and TEQ): Users and uses.

- Dick Bulmer and L. D. Pearlstein (LLNL)
  - ⇨ Vertical stability studies, PF and control system design for ITER and TPX; ITER variants; spheromaks / spherical tokamaks
- Scott Haney (LLNL)
  - ⇨ ITER shutdown scenarios, ITER variants
- Chuck Kessel (PPPL)
  - ⇨ Vertical stability studies for TPX
- Dave Humphreys (GA)
  - ⇨ Control of DIII-D and ITER
- Tom Casper and Barry Stallard (LLNL)
  - ⇨ High beta-p and current ramp scenarios for DIII-D
- G. Tinios and I. H. Hutchinson (MIT)
  - ⇨ Comparing vertical event modeling with Alcator C-MOD
- ITER-Naka team
  - ⇨ ITER modeling

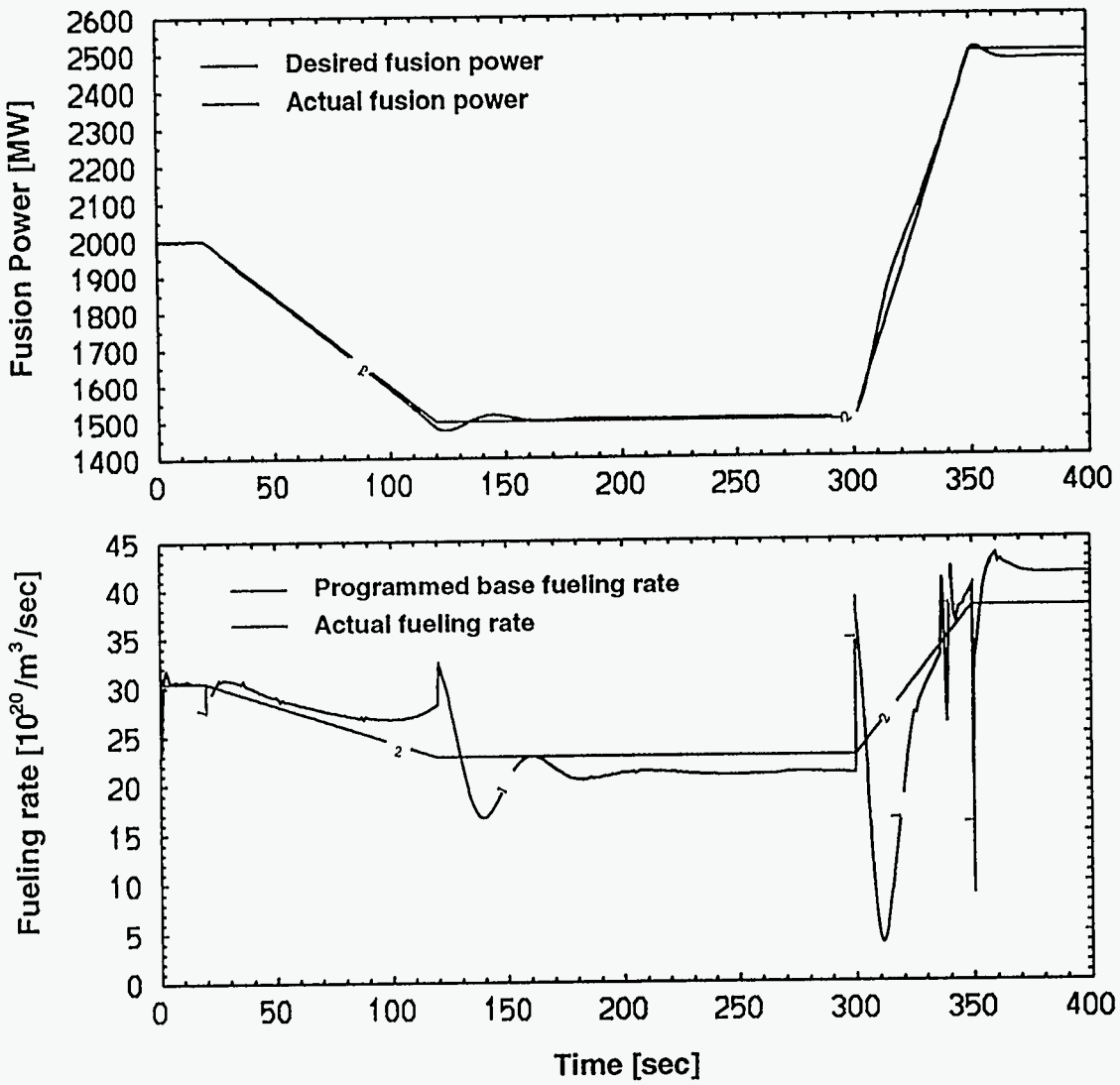
ITER ELM studies.



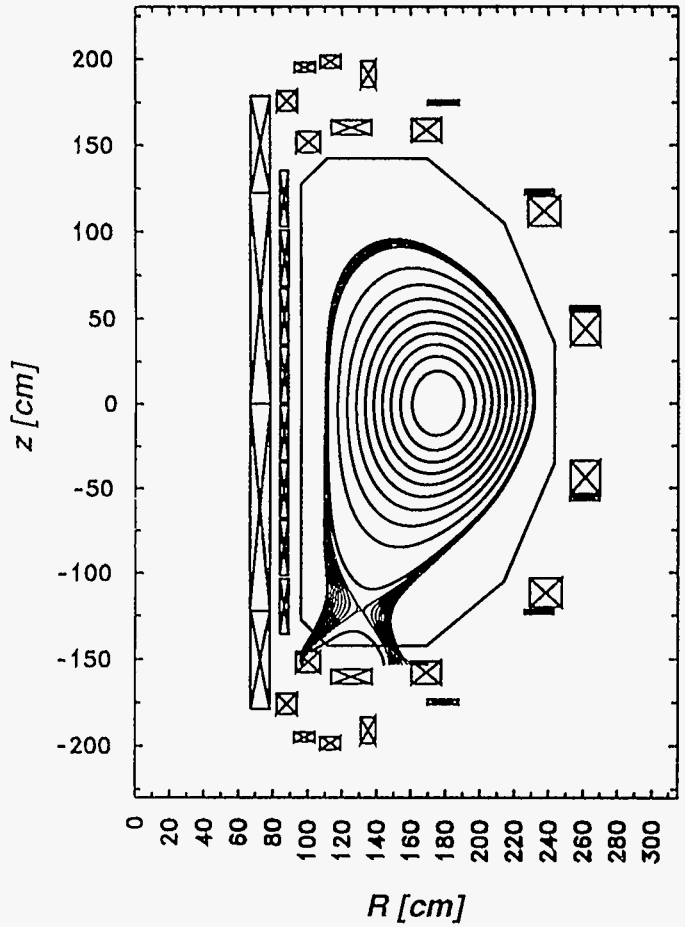


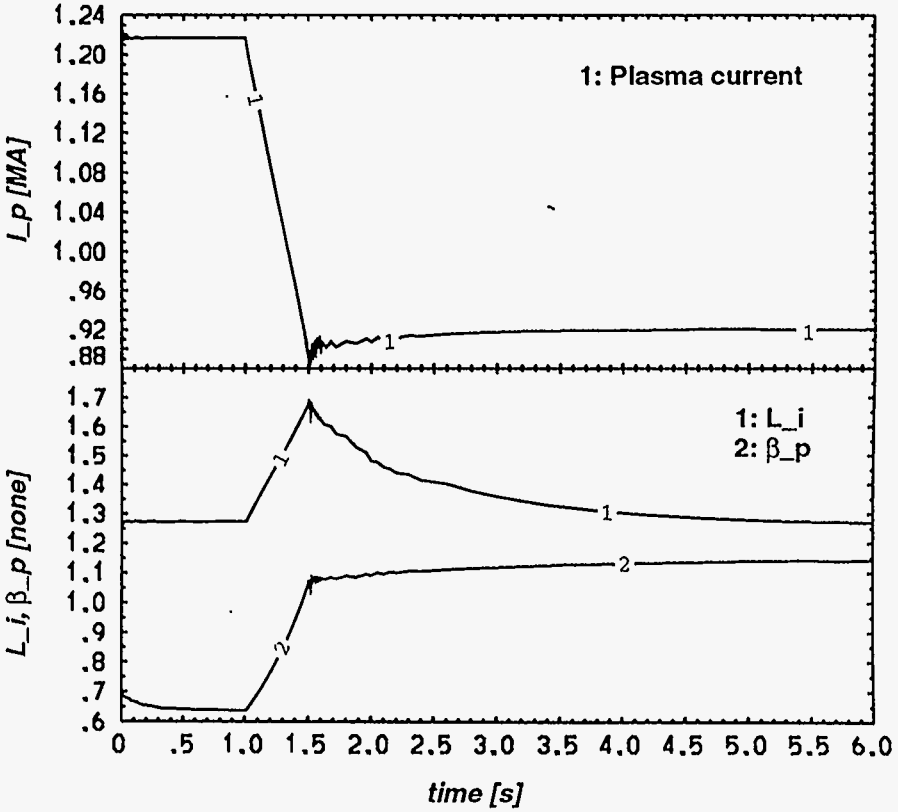


ITER fusion power operating point control example.

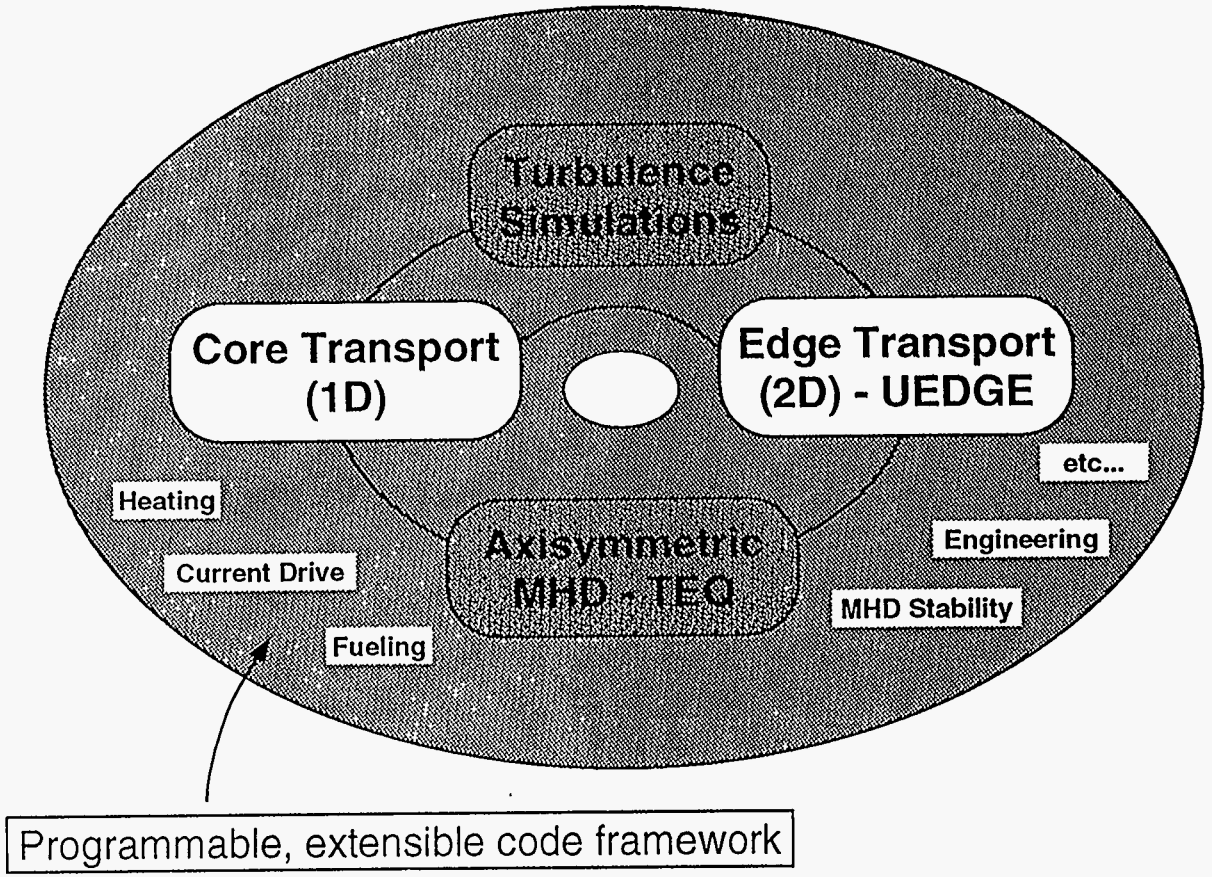


DIII-D current ramp.





CORSICA 2: Coupling core and edge simulations.

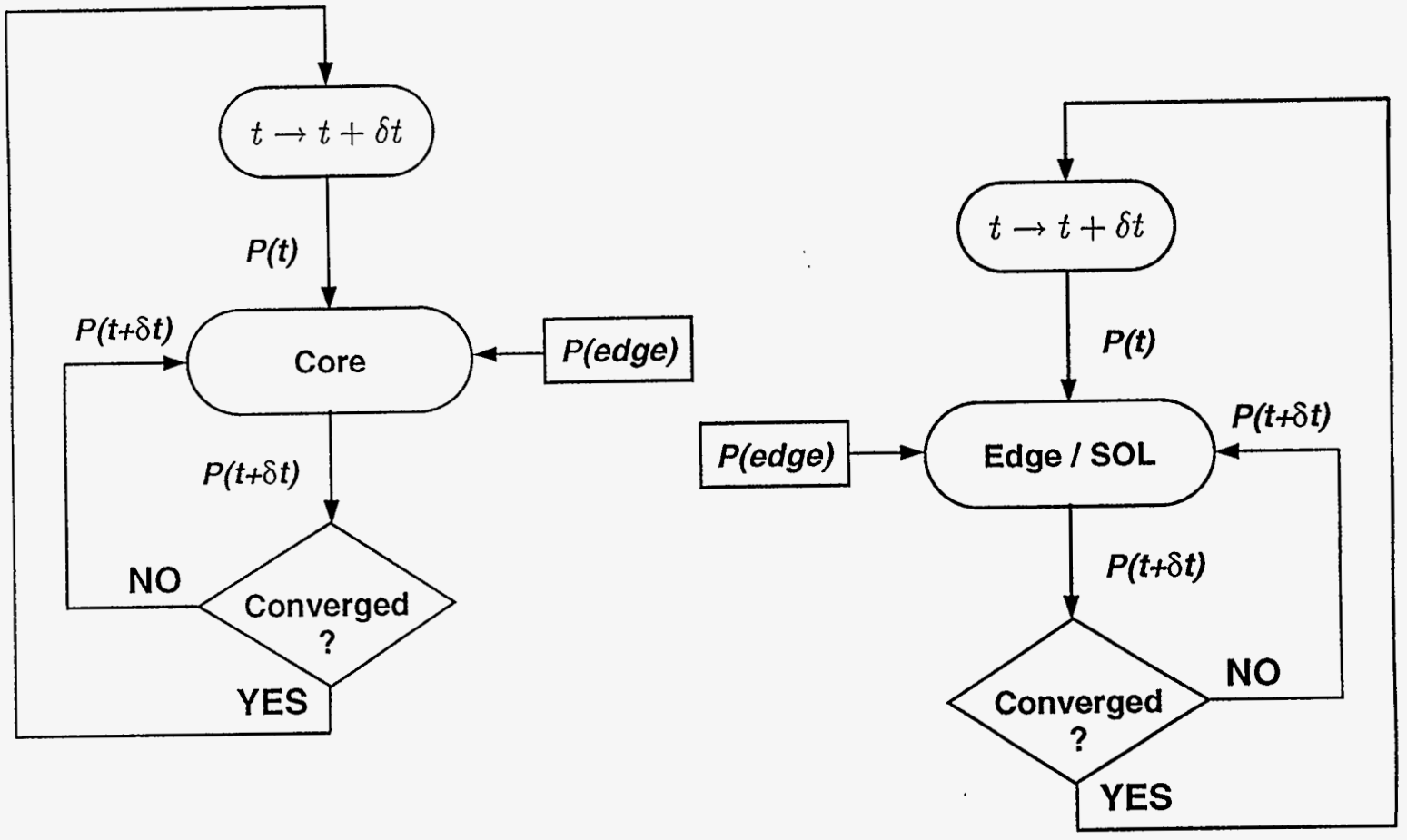


Coupling the core to the edge.

- A big problem with separate core and edge simulations is that the shared boundary condition is unknown. We need the capability to couple a core simulation code to a real 2-D edge code, like UEDGE.
- This coupling is difficult because of the difference in scale and dimensionality between the core and the edge.
- Also, the edge codes are very expensive to run, so it is important that the coupled code be implicit in its timestepping.
- We have developed two such algorithms, tested them in prototype codes, and are now testing them in CORSICA.

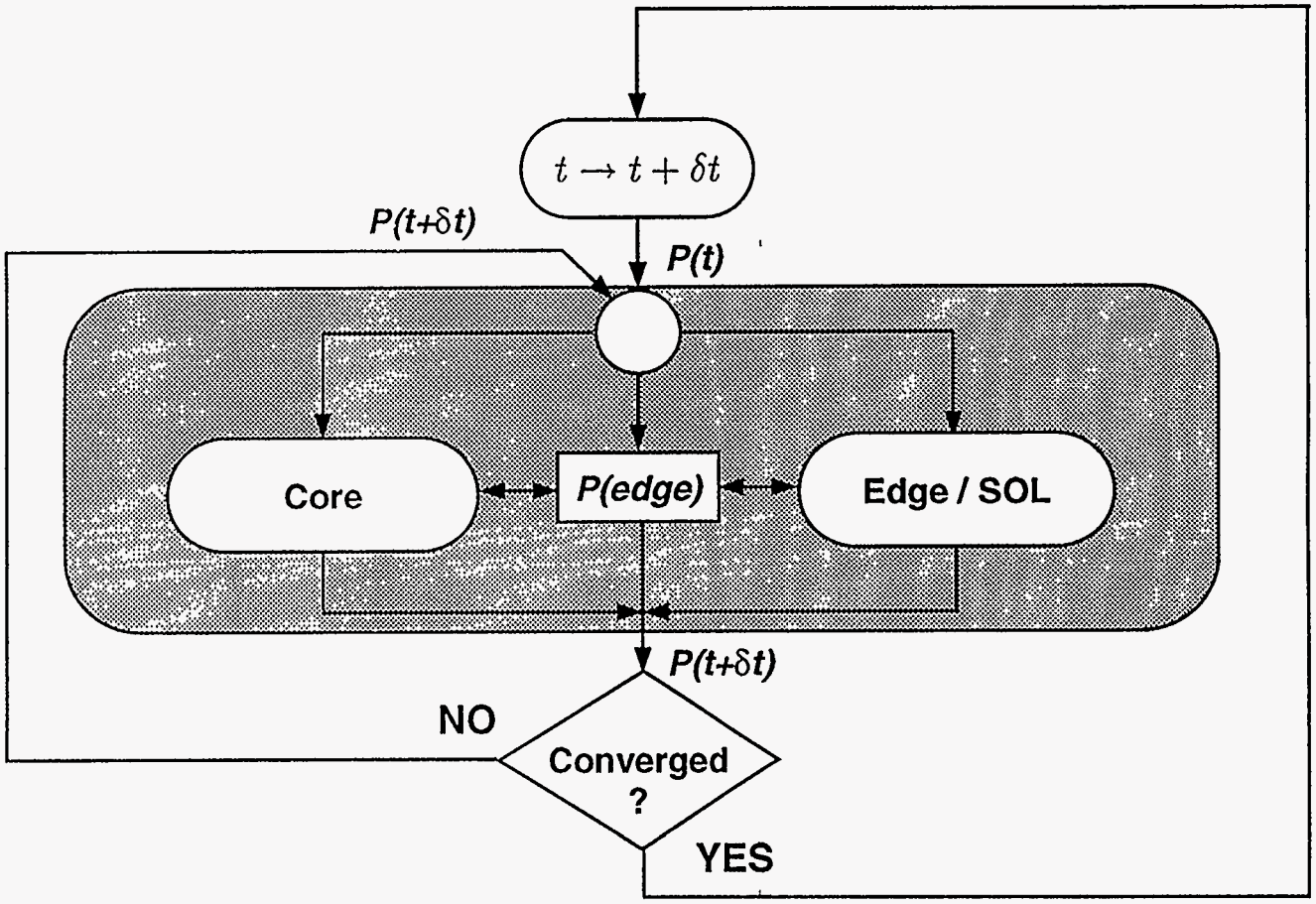
CORSICA 2:
Schematic representation of core and edge codes.

P = Profiles



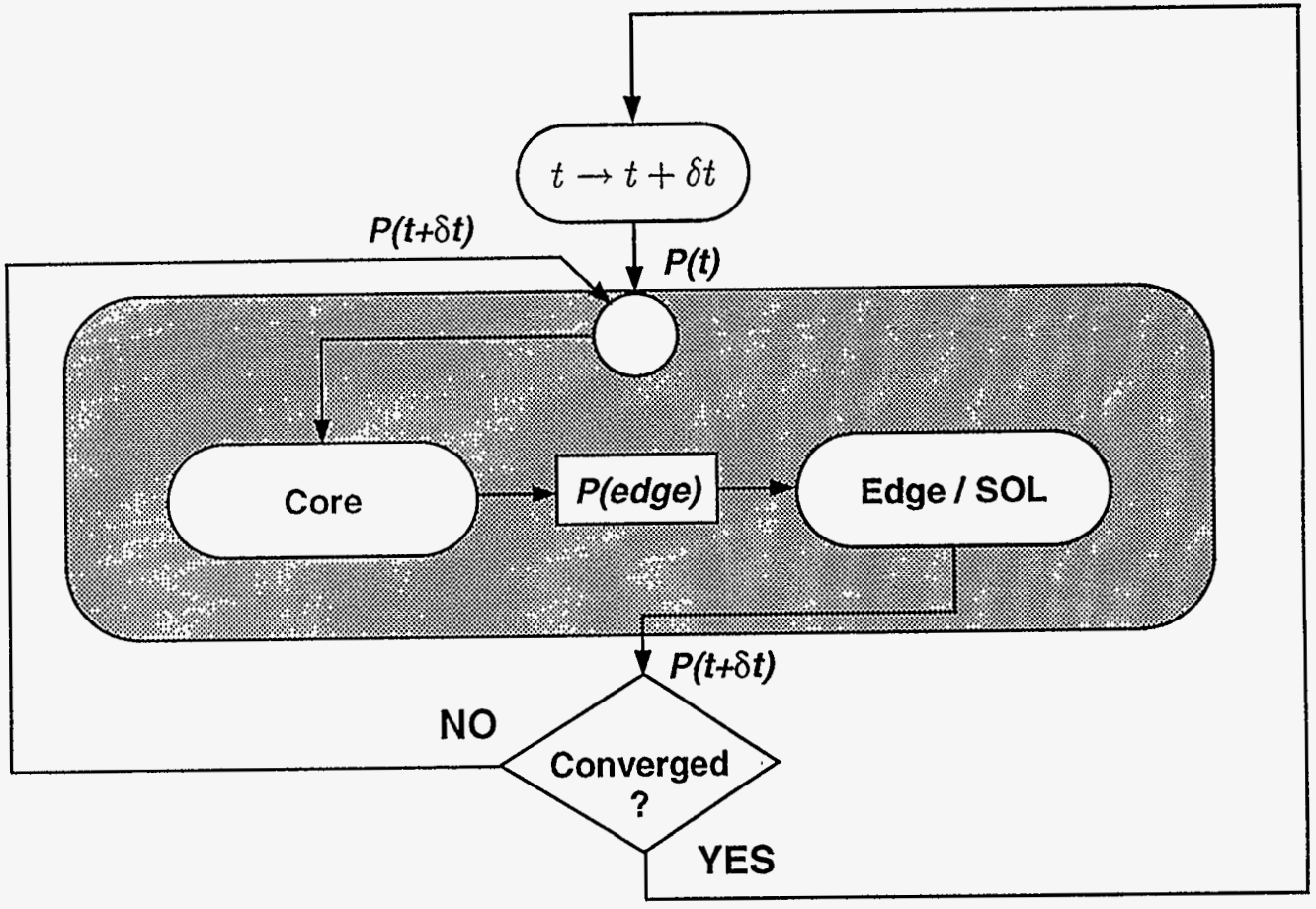
CORSICA 2:
Explicit coupling.

P = Profiles



CORSICA 2: Iterative coupling.

P = Profiles

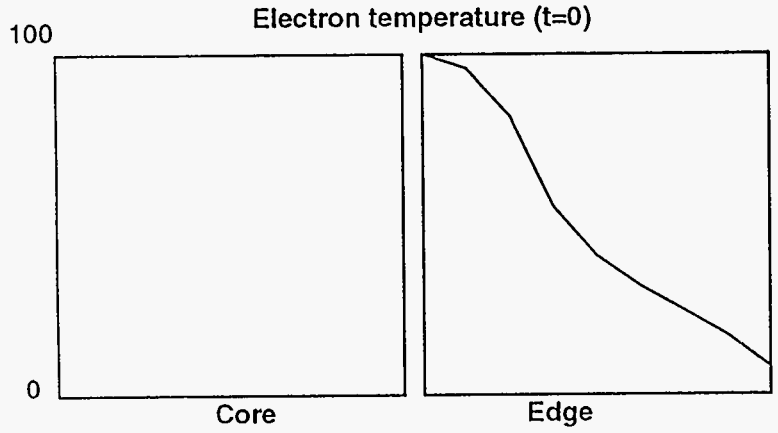


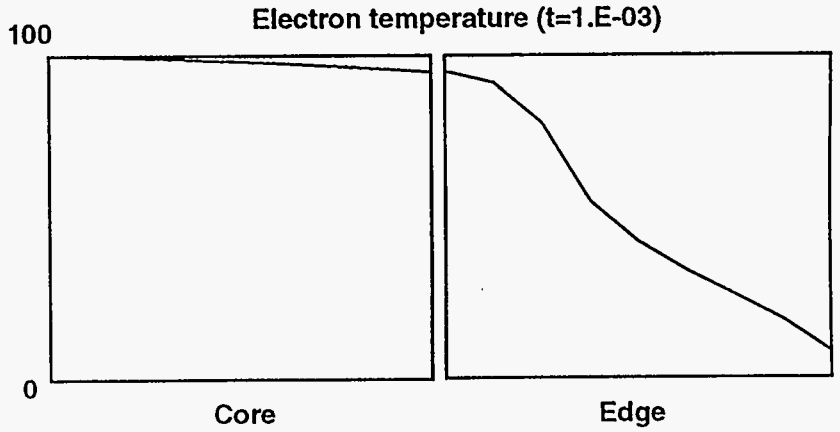
Core-edge coupling status.

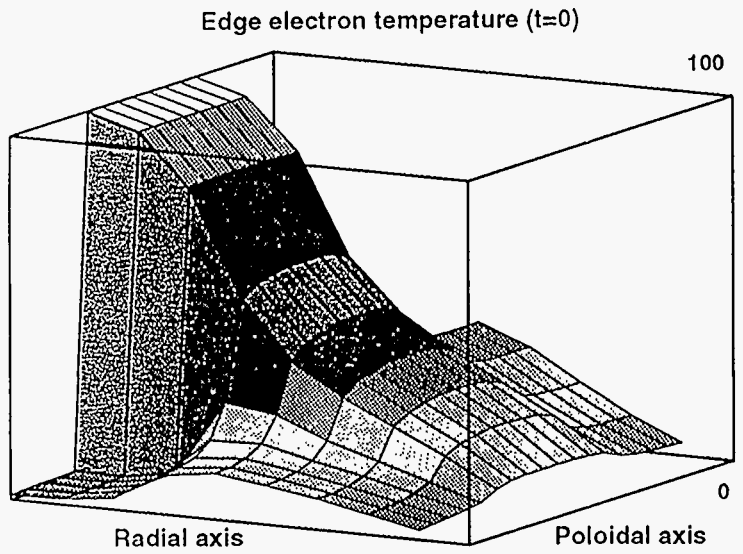
- Algorithms tested in prototypes that coupled 1-D and 2-D slab codes and that coupled a 1-D slab code to the UEDGE edge code.

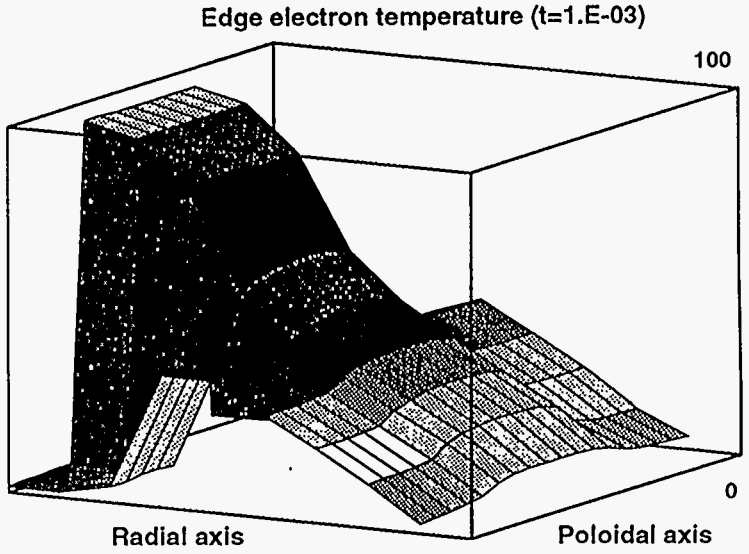
- Recently coupled UEDGE to the CORSICA transport code with fixed MHD equilibrium. This is still in the testing phase.

Core–Edge coupling: Temperature evolution in coupled code.

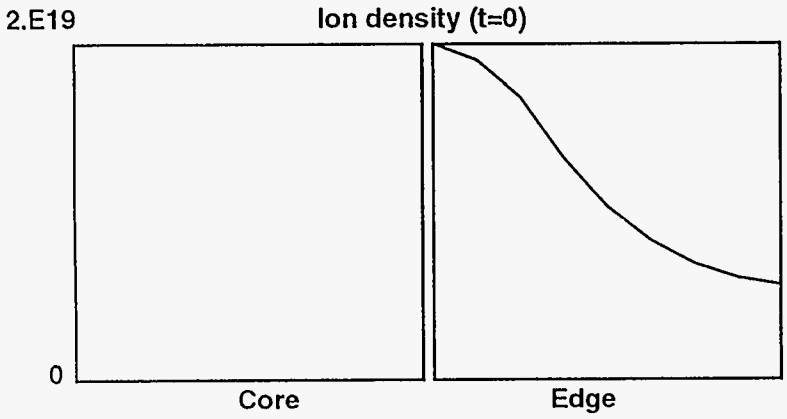


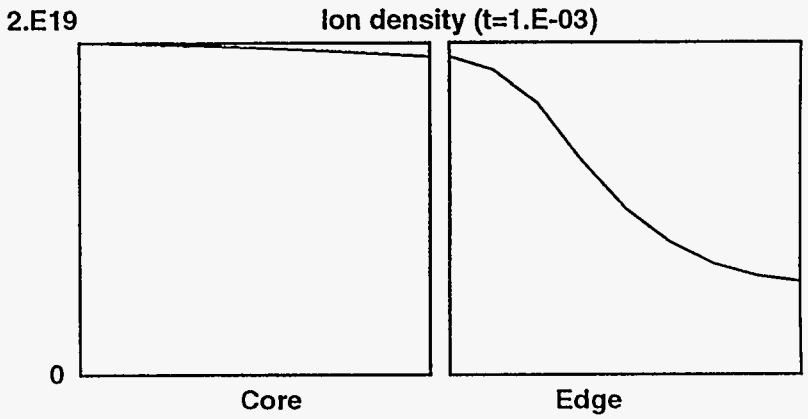


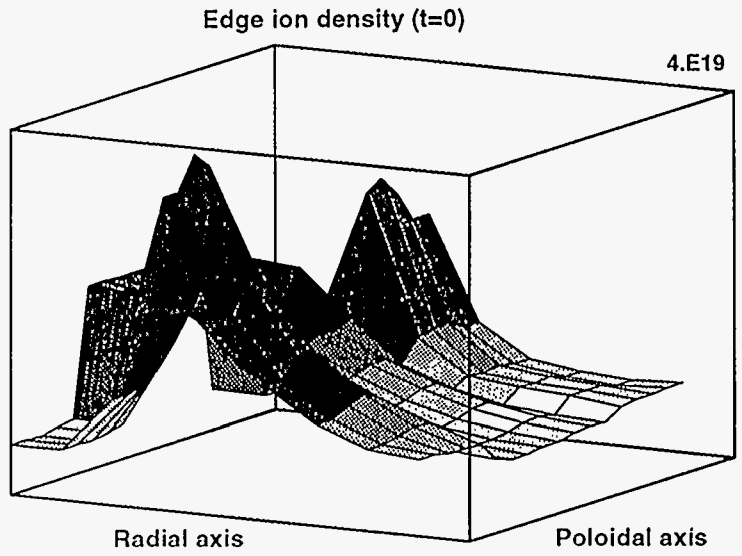


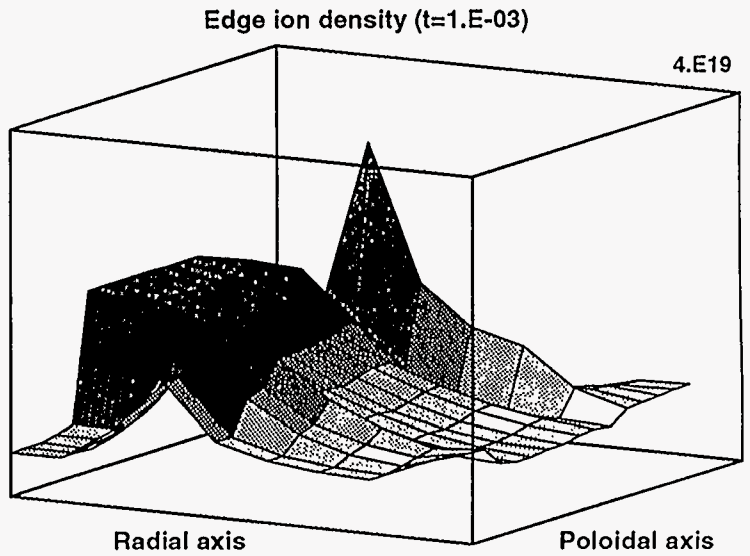


Core–Edge coupling: Density evolution in coupled code.

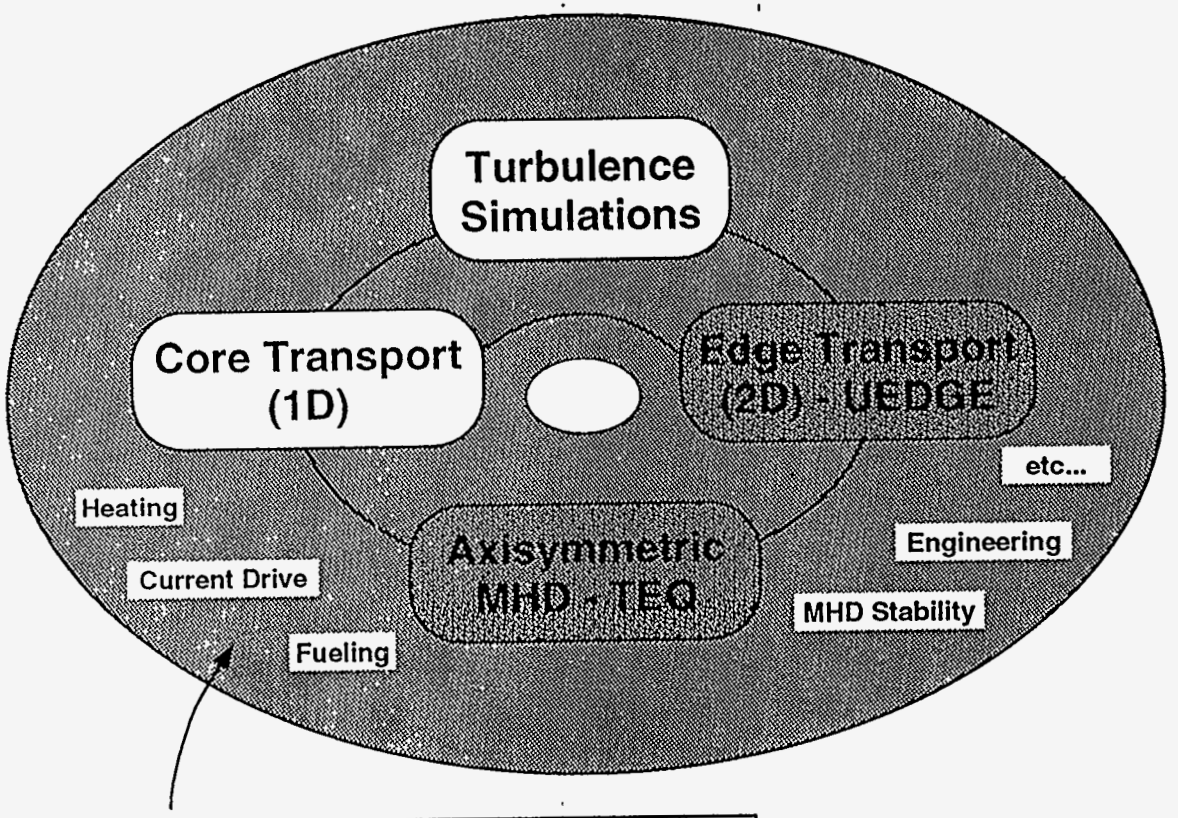








CORSICA 3: Coupling transport to turbulence.

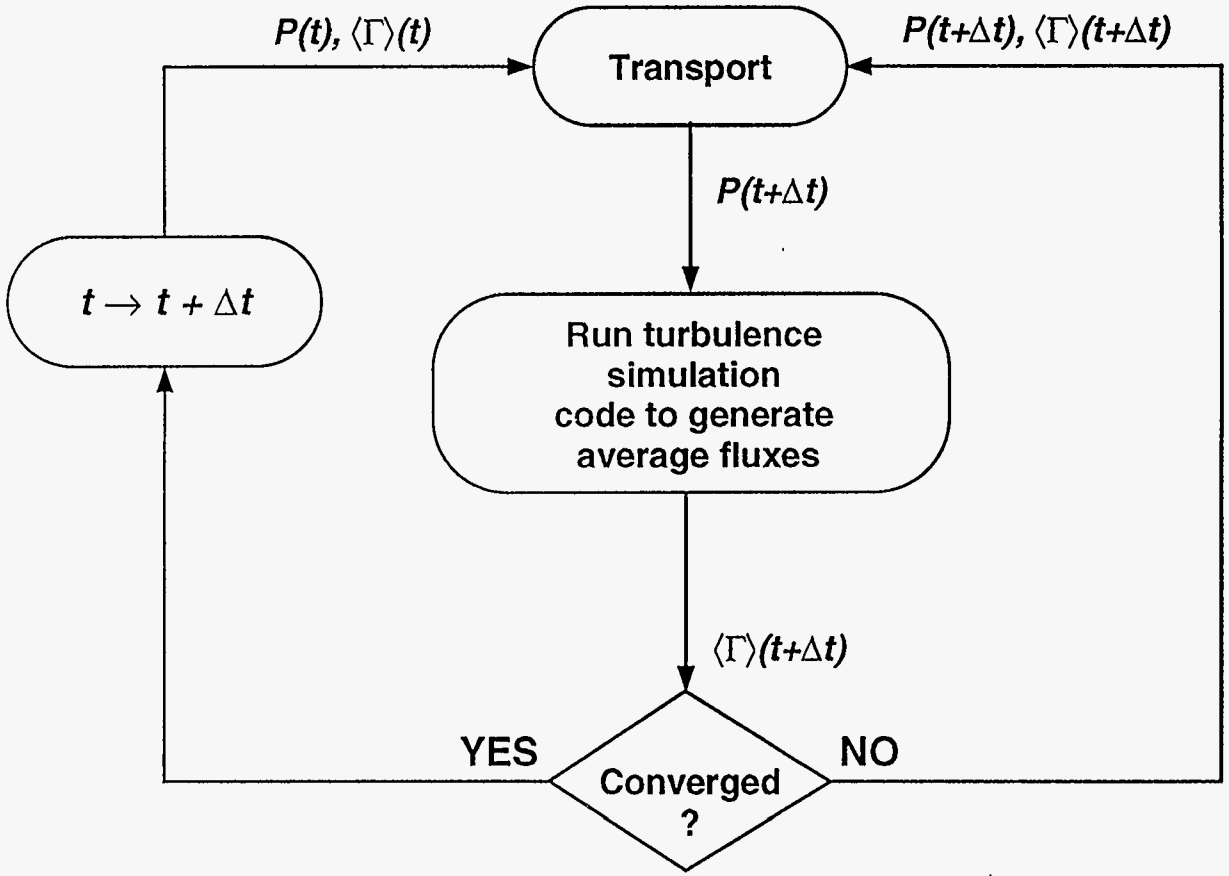


Coupling transport to turbulence.

- Need the ability to do shot-length or steady state simulations including turbulent fluxes that are consistent with the transport profiles.
- Gyrokinetic and gyrofluid turbulence simulations are far too expensive to use as design tools.
- If the timescales for transport and for the underlying turbulent fluctuations are well separated, coupling can be done efficiently.
- The ability to do full-core transport simulations consistent with simulation-generated turbulence fluxes will lead to new insight into transport, and hopefully to improved analytic models.

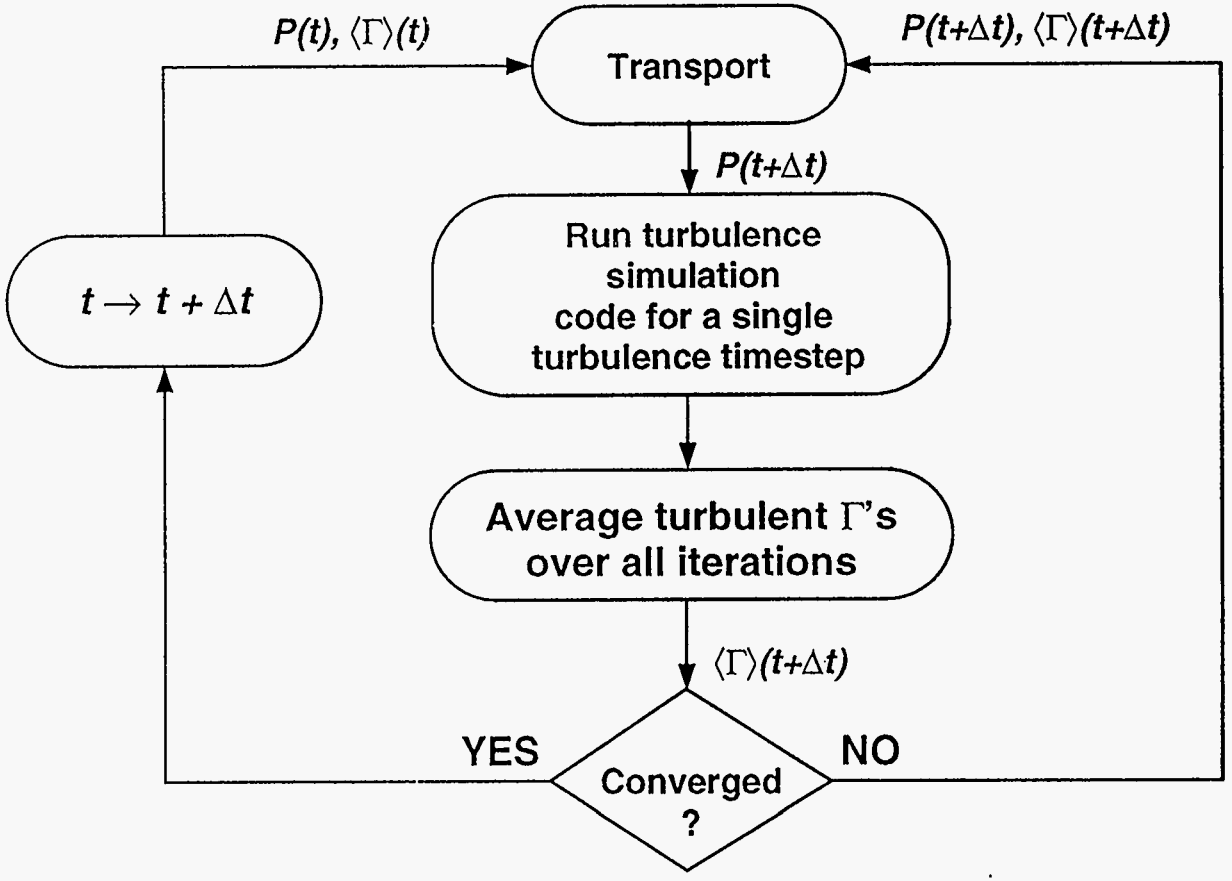
Turbulence-transport coupling: Naive approach.

P = Profiles
Γ = Turbulent fluxes



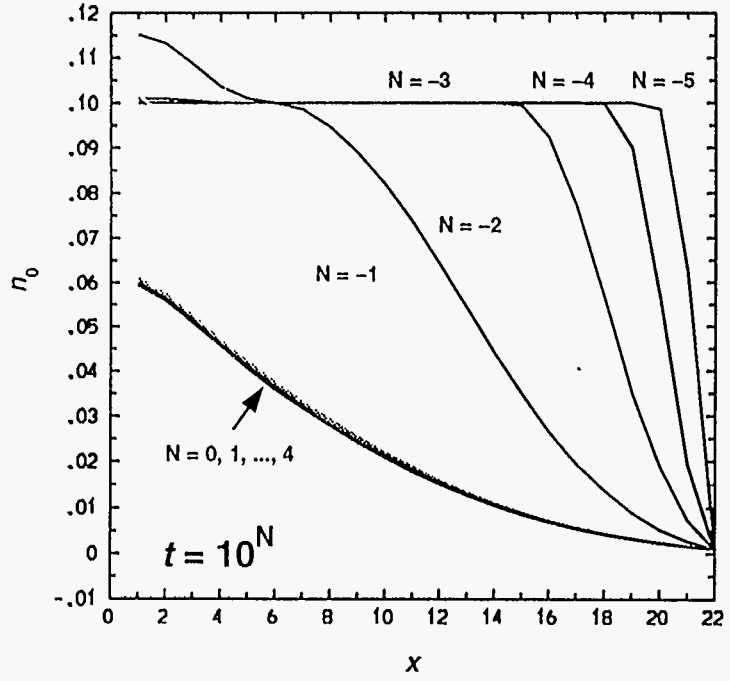
Turbulence-transport coupling: Converge both loops at once.

P = Profiles
Γ = Turbulent fluxes

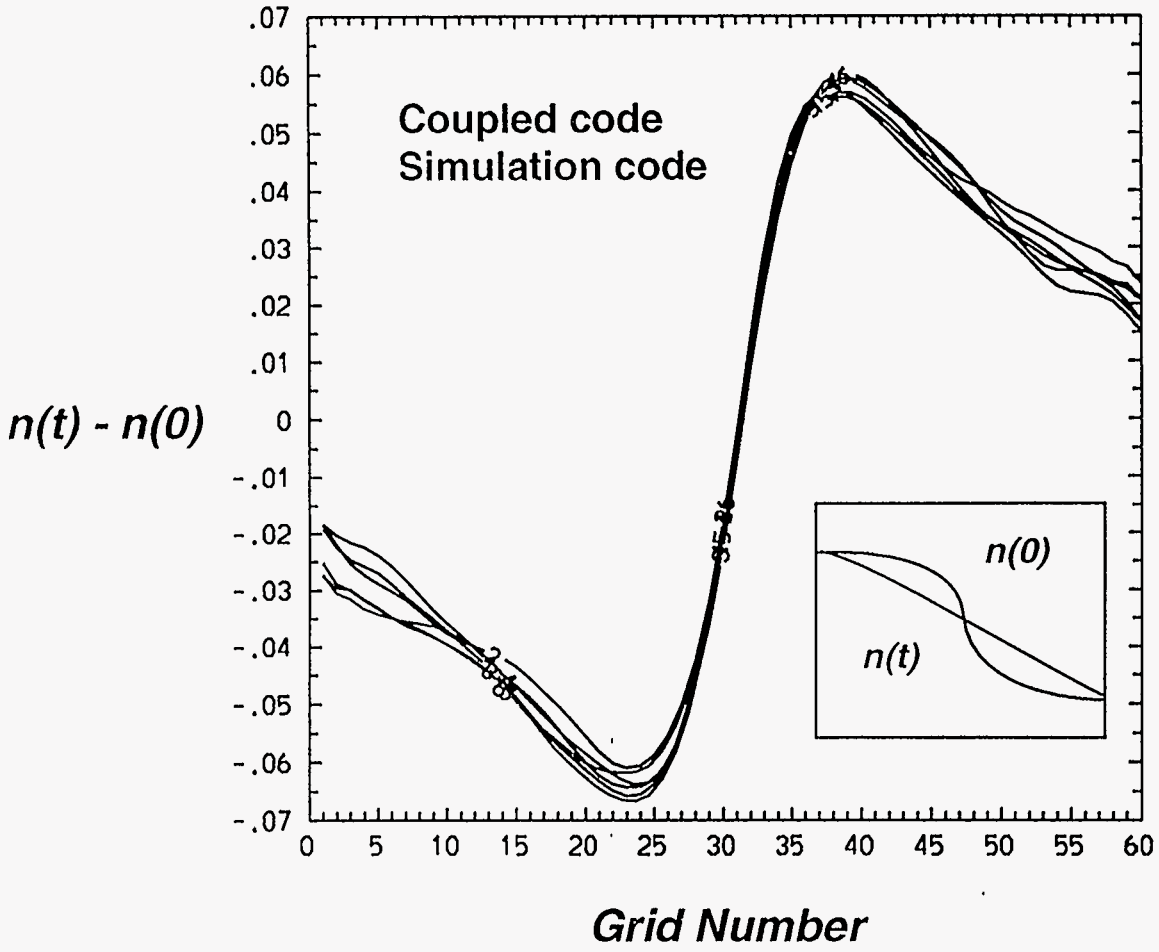


Transport–turbulence coupling: Coupling to simple drift-wave turbulence simulation.

- Transport equation for average density.
- Fluxes generated by Hasegawa-Wakatani drift-wave simulation at each transport grid point.



Transport–turbulence coupling: Coupling to non-local simulation.



Summary.

- CORSICA 1
  - Used for vertical stability, PF and vertical control system design in ITER and TPX,
  - Used for transport modeling of ITER (shutdown, power control, and general operations),
  - Looking at inverse shear modeling of TPX,
  - DIII-D modeling getting underway,
  - Adding shape control capability,
  - Adding inverse equilibrium to make code more robust and to allow it to be used in a fixed boundary mode,
  - Would like to add additional transport models, etc.,
  - Benchmarking against other codes.

- CORSICA 2 and 3 will have many applications, including
  - H-Mode physics,
  - Divertor design,
  - Alpha particle recycling (extremely important to ITER),
  - Self-consistent turbulence simulations in complex geometries with real sources,
  - Test-bed for turbulence models.