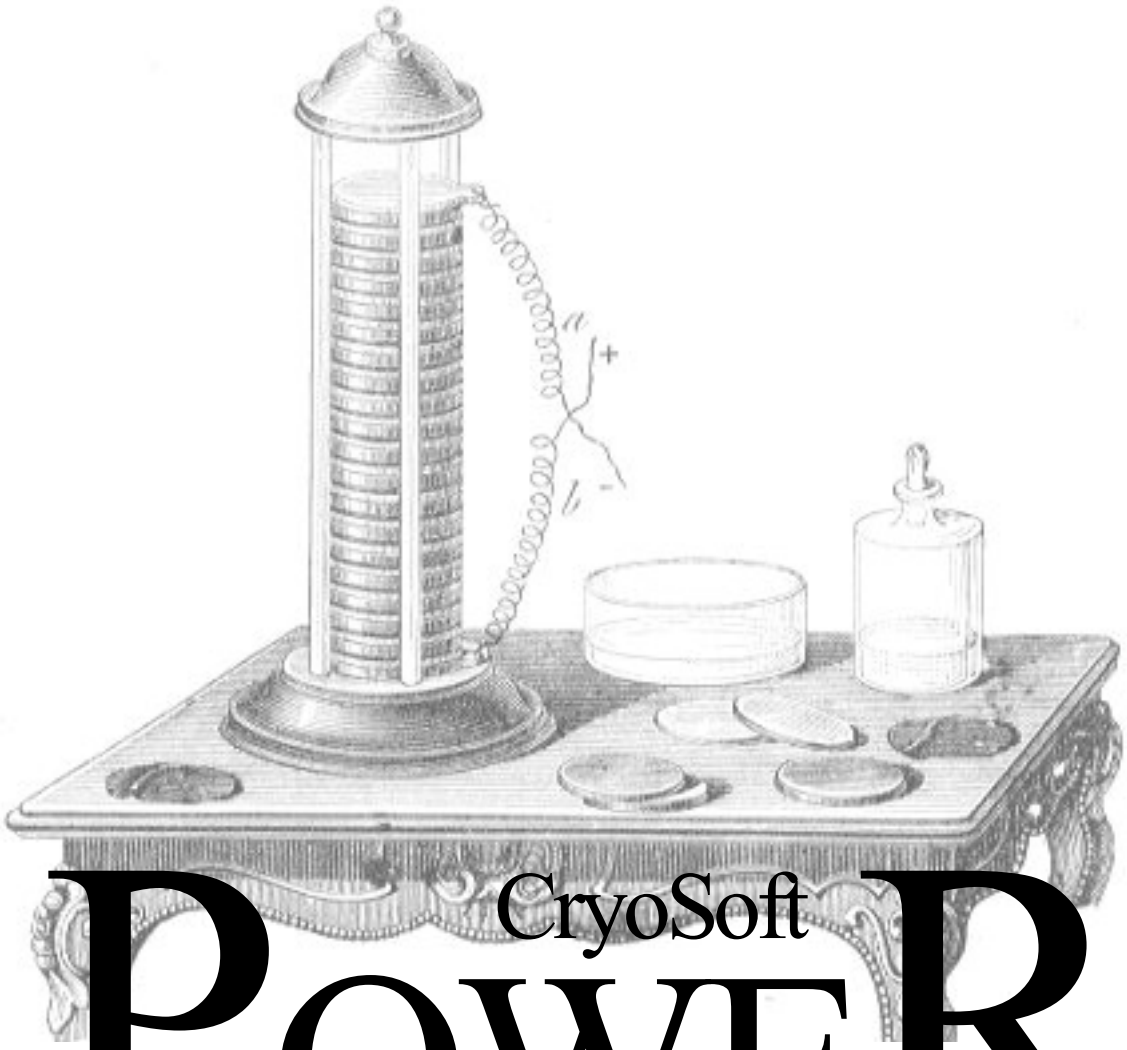
User's Guide

Version 2.1a
November 2021



CryoSoft

POWER

Electric Network Simulation of Magnetic Systems

DISCLAIMER

Even though CryoSoft has carefully reviewed this manual, CRYOSOFT MAKES NO WARRANTY, EITHER EXPRESSED OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS PROVIDED "AS IS", AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL CRYOSOFT BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

Contents

Roadmap

Before you start

This manual is the reference user's guide for POWER and its post-processor, POWERPOST. Throughout this manual we assume that the reader is familiar with the physics and engineering issues that are associated with the design and analysis of an electric network formed of components that can be either passive (linear and non-linear resistors and inductors) or active (linear and non-linear voltage and current sources). For the preparation of the input, running and interpretation of the results of POWER it is mandatory that the user is familiar with the network theorems (Kirchhoff voltage and current laws) as well as with the concepts of *branch* and *mesh*, as standard practice in electrical engineering.

How to use this manual

This manual is structured as follows:

- Chapter 1 contains a brief and general introduction on the modeling principle.
- Chapter 2 gives basic information on the installation, explains how to start a POWER run and launch the post-processor POWERPOST on a UNIX workstation.
- Chapter 3 contains case studies that the reader should use to familiarise with the operation and features of the program.
- Chapter 4 contains additional information on the preparation of the input and the meaning of the input variables
- Chapter 5 describes the details of the post-processing command language.
- Chapter 6 describes the External Routines that can be used for advanced use. These routines can be linked to the standard code to provide powerful customization.
- Chapter 7 deals with troubleshooting and error messages;
- Chapter 8 gives the references and a general bibliography for documentation.

Beginners to POWER should read chapters 1, 2 and 3 in sequence. They will make occasional cross reference to chapters 4 and 5 for detailed information. Experienced users will use chapters 4, 5 and 6 for daily operation. Chapter 7 is designed to be consulted as an indexed glossary for error messages and associated actions.

CHAPTER 1

Introduction

What is POWER

POWER is a program for the simulation of an electric network consisting of resistances, inductances, current and voltage sources, with arbitrary interconnection and coupling. It is specially tailored for the transient analysis of the evolution of current and voltage in a coil connected to a power supply.

A POWER model

In a POWER model we consider a general electric network assembled from lumped parameters components:

- resistances (constant or voltage/current dependent);
- inductances
- voltage sources (constant or current dependent);
- current sources (constant or voltage dependent);

With these hypotheses the network can be solved using a general equation written in the meshes:

$$L\frac{dI}{dt} + RI = V \tag{1}$$

where L and R are the inductance and resistance matrices of the network, I is the array of the mesh currents and V are the mesh external voltages. Note that Eq. (1) is intended as written on meshes assembled by the user using the basic components enumerated above, and specifying the connection of the components (branches) in the mesh. Because POWER is not thought as a general circuit analysis package, the mesh identification is left to the user (through input). An example of an arbitrary assembly of branches (electrical components) in an electrical circuit that can be simulated is shown in Fig. 1. Note that this is not the only circuit that can be modeled with POWER and that the modeling capabilities can be much more extended than this somewhat *classical* circuit used for the protection of a superconducting coil.

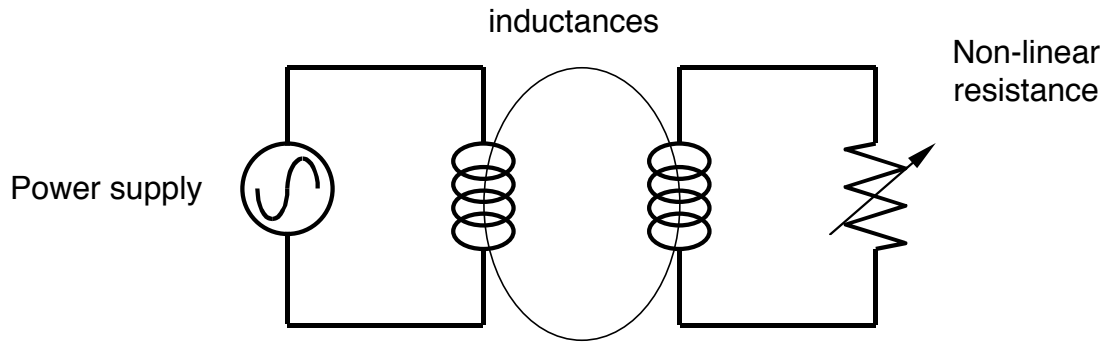


Figure 1 Schematic view of an electric network solved by the network solver POWER. In this example the inductances could represent parts of a coil winding, inductively coupled, and connected to a time-variable power supply. Non-linear resistances can be included in the model, e.g. to represent the behavior during a quench.

Model Solution

The equation for the network of resistances, inductances and sources reported above (see Eq. (1)) is solved in POWER using a fully implicit time stepping algorithm of first order accuracy with fixed time step. This very simple choice has been taken because it produces stable results for most conditions of practical use, and gives to the user direct control of the accuracy of the solution through the specified time step. In case of non-linarites (e.g. if the resistance of a branch is variable in time) the system matrix is built at each time step using the solution at the time reached by the simulation to advance the following time step. No iteration is performed, and again accuracy can be achieved reducing the time step.

Post-processing

The results produced by POWER are integrally stored and can be analyzed to produce plots and reports by the post-processor POWERPOST. POWERPOST responds to a user-friendly command language and allows selection of results in time or space, plot and print-out of results vs. time or space, parametric plot of results at given time or space coordinate. See the case studies in Chapter 3 for examples of post-processing sessions, and Chapter 5 for the details on the syntax of the command language.

User Flexibility and Further Extensions

POWER has several features that allow to customize its modeling capability beyond the allowable parameterization of the thermal/hydraulic/electric configuration that can be achieved using the standard input file. Specifically, the user can:

- modify the dependence of geometry, waveforms and material properties on space, time and solution variables, beyond the standard models implemented, using *External Routines* that can be statically linked to the program segments through a compilation step that produces a customized version of the code. See Chapter 6 for documentation on *External Routines*;
- change parametrically the behavior of the *External Routines* by making use of *Variables* that are read by the code input parser, and can be accessed at run-time using the *Variables*

library. See Chapter 4 for details on the syntax to be adopted for the *Variables* input block;

- couple to other programs of the CryoSoft suite through the multi-tasking code manager SUPERMAGNET. This allows augmenting the physics span of the simulation domain to include thermal networks (e.g. heat exchange in a coil), hydraulic networks (e.g. proximity cryogenics) or electrical circuits (e.g. magnet protection).

CHAPTER 2

Installing and Running POWER

Platforms

POWER and its post-processor POWERPOST are provided as a package developed for running under UNIX or UNIX-like (e.g. Linux) operating system. The reason is that they require computer intensive calculations, orderly file management and little interactivity. At the time when this manual is written, the platform where POWER is developed is:

- Macintosh running MacOS-X (10.10.5 and higher) under XQuartz,(2.7.8) gcc (5.1) with gfortran.

At different time of the development and production, the code has been installed and tested on the following platforms:

- Mac-OS X (10.2 and higher) operating system;
- GNU/Linux operating system (most distributions).
- INTEL PC's running RedHat Linux OS;
- IBM-RISC workstations running the AIX-V4 operating system and later;
- SUN-SPARC workstation running the Solaris OS operating system;
- DEC-ALPHA workstation running the OSF-1 operating system;
- HP workstations running HP-UX OS;
- Windows-2000 and Windows-XP operating system, with an installed CYGWIN environment (the reference version tested is CYGWIN 1.5.24-2).

Although UNIX obeys strict standards, the architecture of the operating and file system may vary from vendor to vendor. It is therefore possible that porting may require minor adaption of code and libraries. Contact us for advice.

In the following sections we assume here that you are running under a UNIX or UNIX-like operating system, and that you are familiar with UNIX commands, directory and file handling. Contact your system administrator for matters regarding UNIX commands and file system.

Although versions of POWER and POWERPOST have been ported to PC's running the Windows OS, at the time when this manual is written this is not a platform directly supported and part of the instructions provided below (i.e. how to run and post-process a case) may not be directly applicable.

Installation

POWER is one of the CryoSoft family of programs. You will have therefore received the CryoSoft package containing POWER either as a tar-ball or in pre-installed form. Verify in the CryoSoft installation manual [1] the procedure to be followed for the proper installation of the complete package. The executable codes, `power` and `powerpost` are in the directory `~/CryoSoft/bin/`. You will find the example inputs and post-processing command files in the directory `~/CryoSoft/xample/power/code_x.x/` (the symbol `~/` stands for your home directory, `x.x` is the version you received).

How to run POWER

Start-up To run POWER you will need to launch the executable code. In the standard installation on a UNIX system described above POWER is launched typing the command:

```
~/CryoSoft/bin/power [-i InputFile] [-v/-s] [-h]
```

Note that command line options are not mandatory (enclosed in brackets, following UNIX documentation standard). The meaning of the options is the following:

`-i, --input`	use `InputFile` to parse the input for the run
`-v, --verbose`	print simulation progress on stdout (default)
`-s, --silent`	no output to stdout
`-h, --help`	print a help message

Once launched, the program decodes the options, if any are given, and checks for the specific operation mode requested. If no input file is provided as an option, then the program prompts the user for the input file name. POWER reads the problem definition from an ASCII file whose structure and content are described in detail in Chapter 4 of this manual. Examples of input files are given in Chapter 3. At this time you will enter the name of a file containing the input for the case to be run (e.g. `file.input`):

```
POWER Enter input file name
file.input
```

POWER then parses the input file, performs checks on consistency, configures the case and starts the simulation. A simulation starts from an initial condition at the starting time and advances in time using the time step selected. At each time step POWER emits a message with the real time reached in the simulation (in s) the time step taken (in s) and the ratio of real time to the total time to be simulated:

```
....
Time : 4.980E-03   Step :  1.000E-05   Time/Tend :   0.98900
Time : 4.990E-03   Step :  1.000E-05   Time/Tend :   0.99900
....
```

until the end of the simulation. When the end time of the simulation is reached POWER prints a message reporting the total CPU time used in the run:

```
Total Cpu [s]:    24.059998
```

Each run of POWER produces:

- a binary storage file containing all results stored at user's specified times. The user can control the name of this file, the default file name is `power.store`;

- a log file containing a report on the case run, run statistics and error messages. The user can control the name of this file; the default file name is `power.log`.

Restart After a succesfull completion of a run it is possible to restart the simulation at the last time stored in the binary storage file and proceed with the time integration. A restart procedure is triggered if the input file read by POWER contains the `Restart` command (see Chapter 3 and 4 for details). Assuming that this is the case for the input file `file.restart`, and the program is launched with no command line options, a restart in our example is obtained launching again POWER:

```
~/CryoSoft/bin/power
POWER Enter input file name
file.restart
```

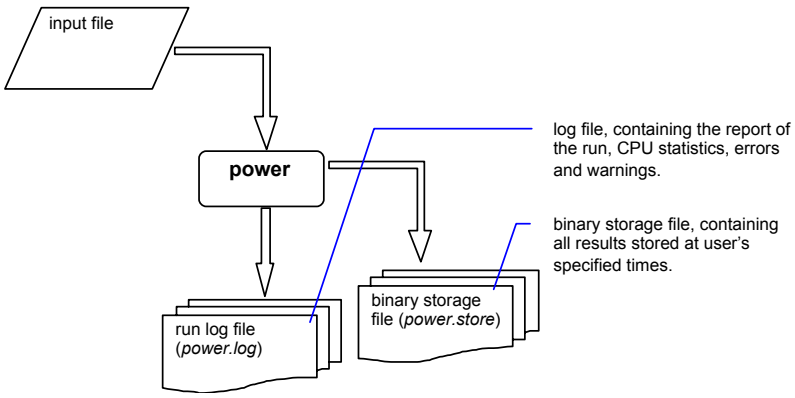
in which case POWER reads the binary storage file and starts the simulation at the last time stored:

```
Time : 5.000E-03   Step :  1.000E-05   Time/Tend :   0.00000
```

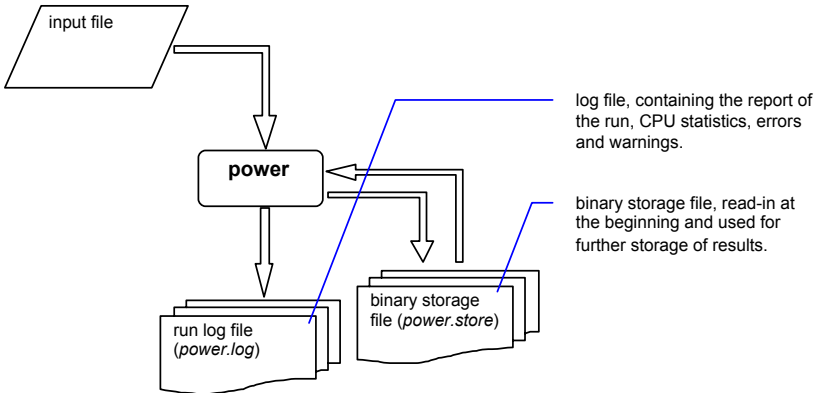
Until the final time specified in the input file `file.restart` is reached.

Note You can use an arbitrary sequence of restarts to simulate different time spans with varying resolution and accuracy. There is no limit to the number of restarts that can be executed for a single simulation.

We show below schematically the flow-diagram of a POWER run:



as compared to the flow-diagram of a POWER restart reported below. Data is read at the beginning of the restart from the binary storage file, and is appended to the same file while the simulation proceeds:



How to run POWERPOST

To produce any detailed result, both in the form of printed tables or plotted curves in PostScript® format, it is necessary to run the POWER post-processor POWERPOST. POWERPOST is launched under UNIX with the command:

```
~/CryoSoft/bin/powerpost [-i InputFile] [-v/-s] [-h]
```

Also in this case command line options are not mandatory (enclosed in brackets, following UNIX documentation standard). The meaning of the options is the following:

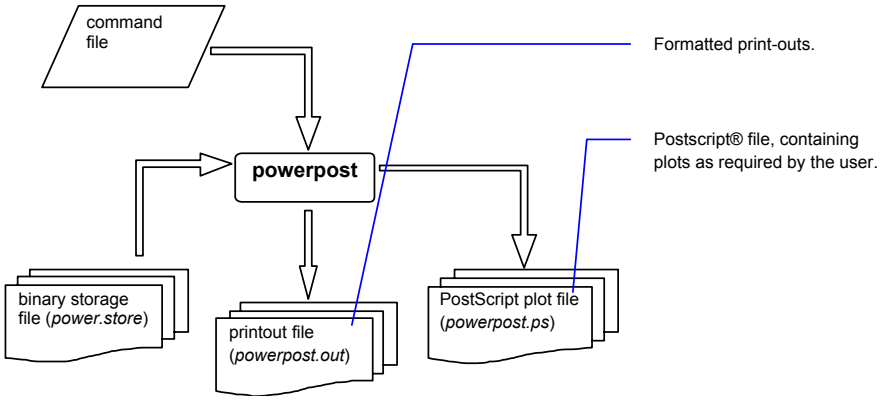
`-i, --input`	use `InputFile` to parse the input for the post-processor
`-v, --verbose`	print post-processing progress on stdout (default)
`-s, --silent`	no output to stdout
`-h, --help`	print a help message

Once launched, the program decodes the options, if any are given, and checks for the specific operation mode requested. If no input file is provided as an option, then the program prompts the user for the name of an ASCII file containing the series of commands that control the generation of the printouts and plots. The structure and content of this file is described in detail in Chapter 5 of this manual. Examples of command files are given in Chapter 3. At this time you will enter the name of the file containing the commands (e.g. `file.post`):

```
Enter command file name
file.post
```

POWERPOST then parses, echoes and interprets the commands from the command file. The commands cause retrieval of the results of a run from the binary storage file generated by POWER (by default from the file `power.store`). As a result POWERPOST generates:

- a file containing the formatted printouts of the results (by default `powerpost.out`), and
- a file containing the plots requested in PostScript® format (by default `powerpost.ps`).



Customization

The method described earlier provides the standard manner to run a POWER simulation, and post-process the results. POWER, however, as most other CryoSoft codes, gives the possibility to customize the physical models by using User Routines, as described in Chapter 6 (see later for details). The user has the possibility to adapt and extend the physics contained in the standard solver, at the additional complexity of writing FORTRAN routines that must obey to the language syntax, and parameter call specification. The customized User Routines need to be compiled and linked the program segments to generate the customized version of the code. Template for the User Routines are given in the directory `~/CryoSoft/usr/power/code_x.x`. Compilation and link-editing can be done using the standard installation script `CSmake`, but we discourage users to modify the standard codes provided, as this will replace the reference installation. As a safer alternative, we strongly recommend copying the User Routines templates in a work directory, and generating in this location the customized version of the code by using an adapted compilation script, or a makefile. Consult the examples below, and contact us for guidelines on how to set-up one such customized structure.

CHAPTER 3

Case Studies

As discussed in Chapter 2, POWER requires an input file with all definitions necessary to specify the assembly of components in the model structure, the characteristics of each component, the initial conditions, and the solution controls. We refer to this file as the *input file*. The input file is needed both for a start-up run and a restart run.

Similarly, post-processing of POWER results using the post-processor POWERPOST requires an input file with a sequence of commands that select results, print and plot them. We refer to this file as the *post-processing command file*.

In this Chapter we give examples of input files and post-processing command files to deal with practical modeling situations. The case studies given here are intended to guide the user from the formulation of a problem to its modeling, the creation of the input file for the case, running the case, and finally the generation of the results. They are simple and are intended as examples to illustrate minimum capability of the program. More complex situations can obviously be modeled, taking the following case studies as starting points and evolving or combining them. Refer to Chapter 2 on how to run the examples described here with POWER and how to generate results and plots with POWERPOST.

Note All input files and post-processing command files for the case studies discussed in this manual are provided with the standard installation. They are located in the directory:

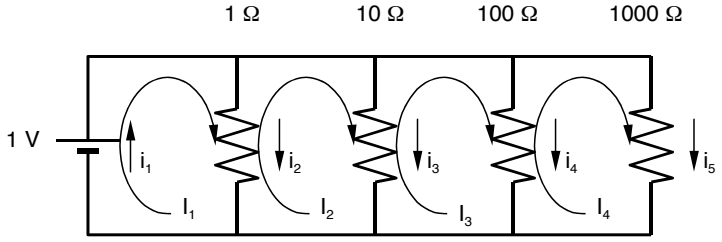
`~/CryoSoft/xample/power/code_x.x`

(where `x.x` stands for the version you received). In the following sections we use the `Courier` font to reproduce the content of those input files, while text in *italic* indicates our comments to the input.

A resistive network

Physical definition of the problem This test illustrates steady state calculation of the current splitting among a set of resistors connected in parallel and powered by a constant voltage source, as shown below. The voltage source provides 1 V, and the resistors have different resistances, 1, 10, 100 and 1000 Ω. The solution is sought at steady state. POWER is a transient simulation, but in this case (with no inductance) the steady state is readily obtained as the solution after a single time step.

The picture below shows the definition of the *branches* of the network (the single components) and the *meshes* (the closed loops formed by the assembled branches). Branch currents are indicated in lower case, mesh currents are indicated in upper case.



The way in which branches are assembled in meshes is defined by the connectivity matrix $C_{i,j}$, a matrix whose rows correspond to the mesh index, and columns to the branch index. The entries in the connectivity matrix $C_{i,j}$ are:

- +1, if the branch j is part of the mesh i and the current direction of mesh and branch is the same;
- -1, if the branch j is part of the mesh i, but the branch current has direction opposite to the mesh current;
- 0 if the branch j is not part of the mesh i.

As an example, the connectivity matrix for the circuit above is given by:

$$C = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix}$$

Input file for the start-up run The step-by-step definition of the input file for the POWER run corresponding to this problem is shown below.

RNet.input

```
      Begin Simulation

Simulation starts at t=0 s and ends at t=1 s, time step is 1 s
            StartTime   0.0
            EndTime     1.0
            TimeStep    1.0

Output is stored (for plotting) at the end time
            OutputStep  1.0

The log and result files are are different from the default ones
```

```
            LogFile     RNet.log
            Storagefile RNet.store
            Title       'resistance network'
```

The test case has 4 meshes formed from 5 branches. This input is needed before the definition of the single branches and meshes to size the memory required

```
            Branches    5
            Meshes      4

      End

      Begin Branch 1
```

Constant voltage source. Entries are in [V]

```
            Type  VoltageSupply   Model constant   Voltage 1.0
      End

      Begin Branch 2
```

Resistance branch, with constant resistance value. Entry is in [Ohm]

```
            Type Resistance   Model constant   Resistance  1.0
      End

      Begin Branch 3
```

Resistance branch, with constant resistance value. Entry is in [Ohm]

```
            Type Resistance   Model constant   Resistance  10.0
      End

      Begin Branch 4
```

Resistance branch, with constant resistance value. Entry is in [Ohm]

```
            Type Resistance   Model constant   Resistance  100.0
      End

      Begin Branch 5
```

Resistance branch, with constant resistance value. Entry is in [Ohm]

```
            Type Resistance   Model constant   Resistance  1000.0
      End

      Begin Mesh 1
```

Definition of the connectivity matrix for mesh 1. This input defines how branches are connected in series to formed a closed mesh. Branches 1 and 2 are connected in series.

```
            Current  0.0   connectivity  1  1  0  0  0
      End

      Begin Mesh 2
```

Definition of the connectivity matrix for mesh 2. Branches 2 and 3 are connected in series. The current in branch 2 flows in negative direction

```
            Current  0.0   connectivity  0 -1  1  0  0
      End
```

```
      Begin Mesh 3
```

Definition of the connectivity matrix for mesh 3. Branches 3 and 4 are connected in series. The current in branch 3 flows in negative direction

```
            Current  0.0   connectivity  0  0 -1  1  0
      End

      Begin Mesh 4
```

Definition of the connectivity matrix for mesh 4. Branches 4 and 5 are connected in series. The current in branch 4 flows in negative direction

```
            Current  0.0   connectivity  0  0  0 -1  1
      End
```

Post-processing command file The following is an example of the sequence of commands necessary to generate of print-outs using the post-processor POWERPOST.

`RNet.post`

Read data from the binary storage file

```
      StorageFile    RNet.store
```

Produce an output file with the name below

```
      OutputFile     RNet.out
```

Currents in each of the branches defined

```
      print current branch 1 branch 2 branch 3 branch 4 branch 5
```

Currents in each of the meshes defined

```
      print current mesh 1 mesh 2 mesh 3 mesh 4

      stop
```

Results One ASCII output file, `RNet.out`, is generated running the post-processor POWERPOST with the commands described abovein the file `RNet.post`. This file contains the output requested. In our case the only output requested are the current in the branches and in the meshes. We report here only an abridged version of the full file.

`RNet.out`

The following is the output of the results. In our case the currents in all branches and meshes defined. The header of the output file, containing an echo of the input, has been removed.

```
POWER Version 2.1
 file created at   30/02/2004   9:58:47
 Storage file:   RNet.store

 Title.........................  resistance network
```

..... (lines omitted)

```
Simulation
 ==========
 number of branches.............          5
 number of meshes...............          4
 Start Time [s].................  0.000E+00
```

```
 End Time [s]...................  0.000E+00
 Time Step [s]..................  1.000E+00


             branch 1      branch 2      branch 3      branch 4      branch 5
Time         Current       Current       Current       Current       Current
[s]          [A]           [A]           [A]           [A]           [A]

-------------------------------------------------------------------------------
  0.0000E+00   0.0000E+00    0.0000E+00    0.0000E+00    0.0000E+00    0.0000E+00
  1.0000E+00   1.1110E+00    1.0000E+00    1.0000E-01    1.0000E-02    1.0000E-03

             mesh 1        mesh 2        mesh 3        mesh 4
Time         Current       Current       Current       Current
[s]          [A]           [A]           [A]           [A]

-----------------------------------------------------------------------
  0.0000E+00   0.0000E+00    0.0000E+00    0.0000E+00    0.0000E+00
  1.0000E+00   1.1110E+00    1.1100E-01    1.1000E-02    1.0000E-03
```

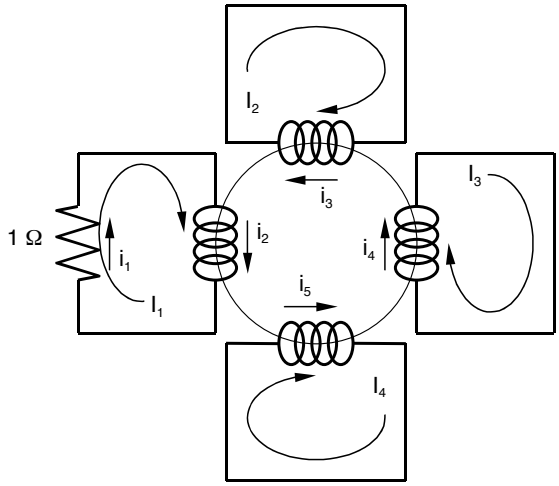
Current decay in a RL network

Physical definition of the problem In this test we demonstrate a simulation of transient current decay in a network composed of passive elements, resistances and inductances. Four circuits are inductively coupled, such as in a superconducting system formed of independently powered magnets. All circuits are initially powered by the same current, 10 A, and in one of them a series resistance of 1 kΩ is suddenly inserted (e.g. a magnet quench). The schematic representation of the circuit is shown below. The inductance matrix for the inductive branches is given by:

$$L = \begin{bmatrix} 1 & 0.4 & 0.1 & 0.4 \\ 0.4 & 1 & 0.4 & 0.1 \\ 0.1 & 0.4 & 1 & 0.4 \\ 0.4 & 0.1 & 0.4 & 1 \end{bmatrix}$$

which simulates stronger coupling among neighbouring inductances. All inductances are given in [H]. The transient behaviour is simulated for a time long enough to show the current decay.

The picture below shows the definition of the *branches* of the network (the single components) and the *meshes* (the closed loops formed by the assembled branches). Branch currents are indicated in lower case, mesh currents are indicated in upper case.



The connectivity matrix for the circuit above is given by:

$$C = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Input file for the start-up run The step-by-step definition of the input file for the POWER run corresponding to this problem is shown below.

`RLDecay.input`

```
Begin Simulation
```

Simulation starts at t=0 s and ends at t=3 s, time step is 5 ms

```
            StartTime    0.0
            EndTime      3.0
            TimeStep     5.0e-3
```

Output is stored (for plotting) every 50 ms

```
            OutputStep  50.0e-3
```

The log and result files are are different from the default ones

```
            LogFile     RLDecay.log
            Storagefile RLDecay.store
            Title       'current decay in coupled R-L circuit'
```

The test case has 4 meshes formed from 5 branches. This input is needed before the definition of the single branches and meshes to size the memory required

```
            Branches    5
            Meshes      4

      End

      Begin Branch 1
```

Resistance branch, with constant resistance value. Entry is in [Ohm]

```
            Type Resistance   Model constant   Resistance  1.0
      End

      Begin Branch 2
```

Inductance branch. Note the inductive coupling of branches 2,3,4 and 5 with the following inductance matrix (compare to entries below and for the other branches:

$$L = \begin{vmatrix} 1.0 & 0.4 & 0.1 & 0.4 \\ 0.4 & 1.0 & 0.4 & 0.1 \\ 0.1 & 0.4 & 1.0 & 0.4 \\ 0.4 & 0.1 & 0.4 & 1.0 \end{vmatrix}$$

Entries are in [H]

```
            Type Inductance
            Inductance  0.0 1.0 0.4 0.1 0.4
      End

      Begin Branch 3
```

Inductance branch. See comments in branch 2

```
            Type Inductance
            Inductance  0.0 0.4 1.0 0.4 0.1
      End

      Begin Branch 4
```

Inductance branch. See comments in branch 2

```
            Type Inductance
            Inductance  0.0 0.1 0.4 1.0 0.4
```

```
End

Begin Branch 5
```

Inductance branch. See comments in branch 2

```
        Type Inductance
        Inductance  0.0 0.4 0.1 0.4 1.0
End

Begin Mesh 1
```

Definition of the connectivity matrix for mesh 1. This input defines how branches are connected in series to formed a closed mesh. Branches 1 and 2 are connected in series. Initial current is 10 [A]

```
        Current  10.0   connectivity  1  1  0  0  0
End

Begin Mesh 2
```

Definition of the connectivity matrix for mesh 2, formed by branch 3 only

```
        Current  10.0   connectivity  0  0  1  0  0
End

Begin Mesh 3
```

Definition of the connectivity matrix for mesh 3, formed by branch 4 only

```
        Current  10.0   connectivity  0  0  0  1  0
End

Begin Mesh 4
```

Definition of the connectivity matrix for mesh 4, formed by branch 5 only

```
        Current  10.0   connectivity  0  0  0  0  1
End
```

Post-processing command file The following is an example of the sequence of commands necessary to generate of print-outs using the post-processor POWERPOST.

`RLDecay.post`

Read data from the binary storage file

```
StorageFile    RLDecay.store
```

Produce a PostScript file with the name below

```
PostScriptFile RLDecay.ps
```

Plot 4 figures on a page, landscape mode

```
set plotsperpage 4
```

Currents in each of the meshes defined

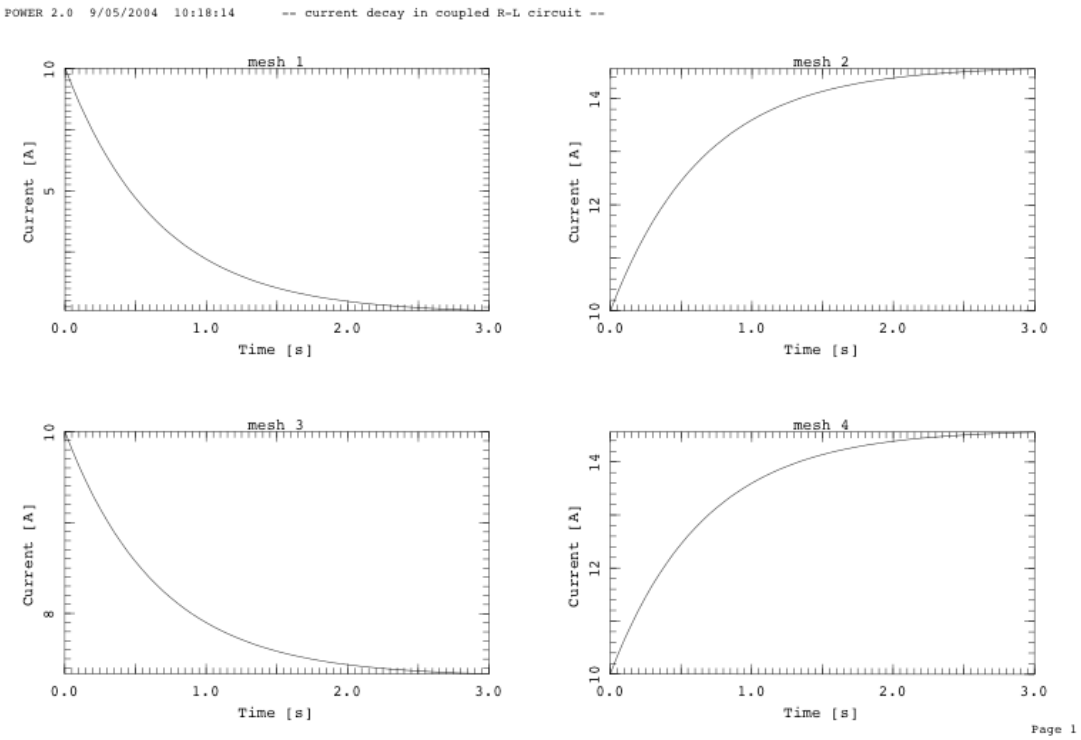
```
plot current mesh 1
plot current mesh 2
plot current mesh 3
plot current mesh 4
```

Voltage in each of the branches defined

```
plot voltage branch 1
plot voltage branch 2
plot voltage branch 3
plot voltage branch 4
plot voltage branch 5

stop
```

Results A PostScript file, `RLDecay.ps`, is generated running the post-processor POWERPOST with the commands described above. The first of the three plot pages generated is shown below, reporting the current in the four meshes. The current in the first mesh, the R-L circuit, decays, while the current in the strongly inductively coupled meshes 2 and 4 increases. Note that because of the choice of the inductances in the matrix , also the current in mesh 3, weakly coupled with mesh 1, decreases as a result.



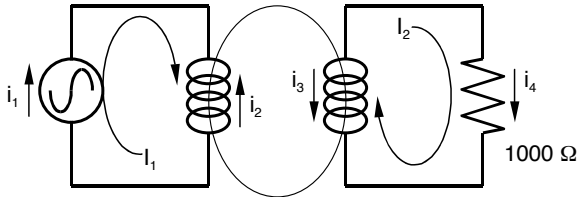
Current driven RL network

Physical definition of the problem In this test we analyse the response of an R-L circuit to an harmonic variation of the current. To this aim we need to define a time-variable current supply, which we realise through a user's defined routine. The network consists of two circuits, inductively coupled, of which one has a current supply that produces a 50 Hz current waveform, with amplitude 10 A, and the second has a series resistance of 1 kΩ. The schematic representation of the circuit is shown below. The inductance matrix for the inductive branches is given by:

$$L = \begin{bmatrix} 1 & 0.8 \\ 0.8 & 1 \end{bmatrix}$$

All inductances are given in [H]. The transient behaviour is simulated for a time long enough to reach the periodic regime.

The picture below shows the definition of the *branches* of the network (the single components) and the *meshes* (the closed loops formed by the assembled branches). Branch currents are indicated in lower case, mesh currents are indicated in upper case.



The connectivity matrix for the circuit above is given by:

$$C = \begin{bmatrix} 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \end{bmatrix}$$

Input file for the start-up run The step-by-step definition of the input file for the POWER run corresponding to this problem is shown below.

RLNet.input

```
      Begin Simulation

Simulation starts at t=0 s and ends at t=50 ms, time step is 0.1 ms
              StartTime    0.0
              EndTime     50.0e-3
              TimeStep     0.1e-3

Output is stored (for plotting) every 1 ms
              OutputStep   1.0e-3

The log and result files are are different from the default ones
              LogFile      test3.log
              Storagefile  test3.store
              Title        'coupled R-L circuit'

The test case has 2 meshes formed from 4 branches. This input is needed
before the definition of the single branches and meshes to size the
```

memory required

```
            Branches    4
            Meshes      2

      End

      Begin Branch 1
```

Current supply branch, providing a current given by the user defined function UserBranchCurrent. The definition there is of sinusoidal current excitation with 50 Hz frequency (see routine)

```
            Type  CurrentSupply
            Model user
```

This value of the current is passed to the user routine as a scale factor

```
            Current  10.0

      End

      Begin Branch 2
```

Inductance branch. Note that the inductance matrix with all other branches is required, although some of the other branches may have no inductance (e.g. resistance or current supply branches). Entries are in [H]

```
            Type Inductance
            Inductance  0.0 1.0 0.8 0.0
      End

      Begin Branch 3
```

Inductance branch. Note the inductive coupling of branches 2 and 3, with the following inductance matrix (compare to entries below and for branch 2):

$$L = \begin{vmatrix} 1.0 & 0.8 \\ 0.8 & 1.0 \end{vmatrix}$$

```
            Type Inductance
            Inductance  0.0 0.8 1.0 0.0
      End

      Begin Branch 4
```

Resistance branch, with constant resistance value. Entry is in [Ohm]

```
            Type Resistance
            Model constant
            Resistance  1000.0
      End

      Begin Mesh 1
```

Definition of the connectivity matrix for mesh 1. This input defines how branches are connected in series to formed a closed mesh. Branches 1 and 2 are connected in series. The current in branch 2 flows in negative direction

```
            Current  0.0
            connectivity  1 -1  0  0
      End
```

```
      Begin Mesh 2
```

Definition of the connectivity matrix for mesh 2. Branches 3 and 4 are connected in series. The current in branch 3 flows in negative direction

```
            Current  0.0
            connectivity  0  0 -1  1
      End
```

User's defined routine As from the input file listed above, the current supply waveform is provided by a user's defined routine, userBranchCurrent (see Chapter 6 for the calling convention). This routine shall be written by the user, using the prototype provided, compiled and linked to produce a customized version of the POWER simulator. The run can only be performed using with customized version, and it is mandatory that the user exerts a maximum of controls on the routine provided as well as a careful management of compiled objects. The listing below reports the routine for the case examined here.

userBranchCurrentRLNet.f

```
c ######################################################################
      real function UserBranchCurrent(id,c0,vtheta,cn,time)
c ######################################################################
c #
c # user's routine for non-linear current generator branch for the test
c # case as from file test3.input
c #
      implicit none
c *
      integer  id
      real     c0,vtheta,cn,time
c *
      real     Omega

c * force the current as I(t) = c0 * sin (Omega * time), where c0 is
c * the generator current as read from input, and Omega is set in the
c * routine to a constant, so that the frequency of the generator is
c * f = 50 Hz

c * set the value of Omega
      Omega = 2.0 * 3.141592654 * 50.0

c * make sure the call to the user routines is for the first branch
c * as defined in input (see input file test3.input)
      if(id.eq.1) then

c * compute current
         UserBranchCurrent = c0 * sin (Omega*time)

      else

         write(6,*) 'userBranchCurrent called with wrong id:',id
         UserBranchCurrent = 0.0

      endif

c *
      return
      end
```

Post-processing command file The following is an example of the sequence of commands necessary to generate of print-outs using the post-processor POWERPOST.

RLNet.post

Read data from the binary storage file
```
StorageFile    test3.store
```

Produce a PostScript file with the name below
```
PostScriptFile test3.ps
```

Produce an output file with the name below
```
OutputFile test3.out
```

Plot 4 figures on a page, landscape mode
```
set plotsperpage 4
```

Currents in each of the branches defined
```
plot current branch 1
plot current branch 2
plot current branch 3
plot current branch 4
```

Voltage in each of the branches defined
```
plot voltage branch 1
plot voltage branch 2
plot voltage branch 3
plot voltage branch 4
```

dI/dt in each of the branches defined
```
plot currentderivative branch 1
plot currentderivative branch 2
plot currentderivative branch 3
plot currentderivative branch 4
```

dV/dt in each of the branches defined
```
plot voltagederivative branch 1
plot voltagederivative branch 2
plot voltagederivative branch 3
plot voltagederivative branch 4
```

Ask for closing the present plot page, and open a new one
```
newpage
```

Plot 2 figures per page, landscape mode
```
set plotsperpage 2
```

Current in the two meshes
```
plot current mesh 1
plot current mesh 2
```

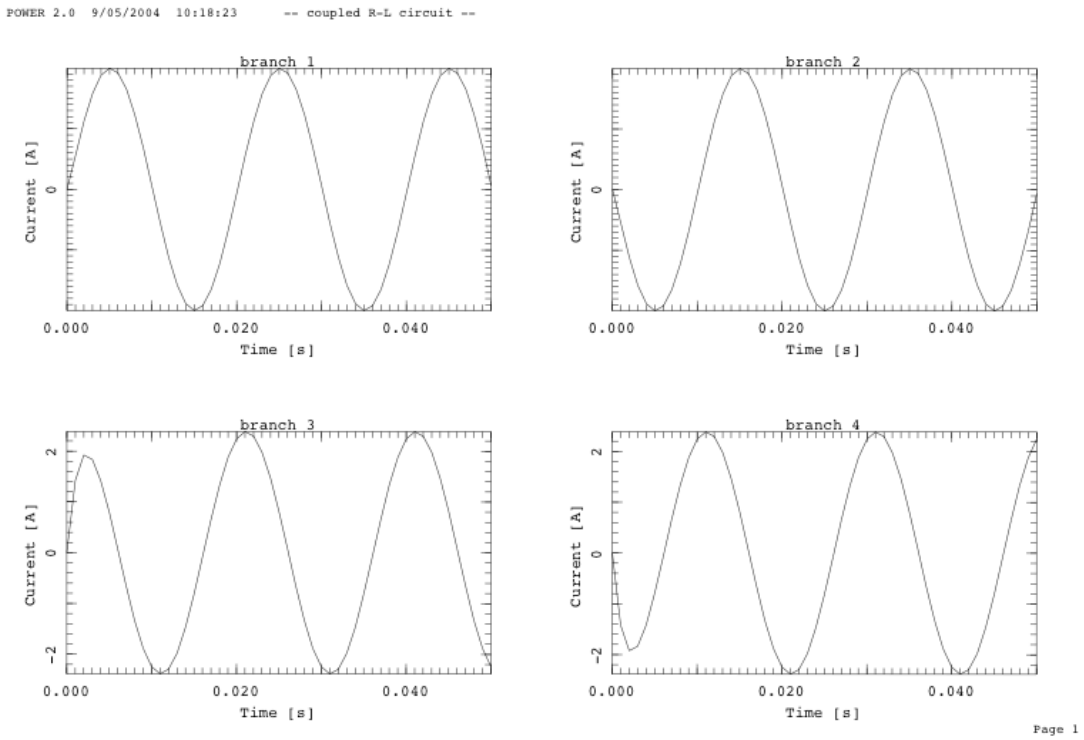
dI/dt in the two meshes
```
plot currentderivative mesh 1
plot currentderivative mesh 2
```

Print a table of currents in the two meshes
```
print current mesh 1 mesh 2

stop
```

Results A PostScript file, `RLNet.ps`, and an ASCII output file, `RLNet.out` are generated running the post-processor POWERPOST with the commands described above. The first of the plot pages generated is shown below, reporting the current in the four branches.

Note that as defined by the Connectivity matrix the currents are equal and opposite in each couple of branches of a mesh. After an initial transient the current reaches periodic regime.



The file `RLNet.out` contains the output requested. In our case the only output requested are the current in the meshes as a function of time. We report here only an abridged version of the full file.

`RNet.out`

The following is the output of the results. In our case the currents in the two meshes. The header of the output file, containing an echo of the input, has been removed.

```
POWER Version 2.1
 file created at    9/05/2004  10:18:23
 Storage file:   test3.store

 Title.........................  coupled R-L circuit
```

..... *(lines omitted)*

```
Simulation
 ==========
 number of branches.............          4
 number of meshes...............          2
 Start Time [s].................  0.000E+00
 End Time [s]...................  0.000E+00
 Time Step [s]..................  1.000E-04


             mesh 1        mesh 2
Time         Current       Current
```

```
[s]          [A]           [A]

-----------------------------------------
  0.0000E+00   0.0000E+00    0.0000E+00
  1.0000E-03   2.7899E+00   -1.4241E+00
  2.0000E-03   5.6208E+00   -1.9245E+00
  3.0000E-03   7.9015E+00   -1.8373E+00
  4.0000E-03   9.4088E+00   -1.4163E+00
  5.0000E-03   9.9951E+00   -7.9739E-01
```

..... *(lines omitted)*

```
  4.9000E-02   3.3875E+00    1.8795E+00
  5.0000E-02   3.1335E-01    2.2428E+00
```

CHAPTER 4

Input Reference

Structure and syntax

The input file is read by the input interpreter that parses and analyzes the syntax and the grammar of the various entries. In general the file contains a series of blocks that are structured as follows:

```
Begin BlockName
        VariableName    value(s)
        VariableName    value(s)
        ……..

        VariableName    value(s)
End
```

where *BlockName* is a keyword indicating the block type, and must be one of the following valid choices:

`Branch`	define the general properties of the branches
`Mesh`	define the general properties of the meshes
`Simulation`	define the simulation parameters
`Variables`	define user variables for use in routines and functions

The content of a block is a series of assignations of a set of values to a generic variable *VariableName*. *VariableName* must be chosen among the set of keywords described in the following sections.

The structure and content of the input file must comply with the following rules and conventions:

- the identifier of a variable and the corresponding value(s) can appear at any position on the line, they can carry on to several lines and must be separated by blanks or tabs;
- the interpretation is case insensitive;
- abbreviations of the keys are not allowed;
- a character ';' in any position of the command line indicates that the remainder of the line must be considered as a comment. If the ';' is the first character in a line, then the whole line is ignored;
- the variables in the block are read sequentially and are checked at read-in time. For this reason the order of precedence of the variables must be respected whenever a value is needed to proceed with the interpretation of a block (i.e. the total number of branches or

meshes must be available before reading the single branch/mesh blocks) . The same BlockName can appear more than once in a file;
- repeated variable assignation overrides previous values and is not checked at read-in time;
- the blocks in the file are read sequentially and are checked at read-in time. The same BlockName can appear more than once in a file

Parsing of the input file is finished as soon as an end-of-file is found. At this point the execution control is passed to the main program that executes checks on data consistency, configures the run and launches the simulation. For sample input files see Chapter 3.

Input variables reference

The following table contains, in alphabetical order, the keywords defining the input variables, their physical dimensions and meanings for each block type. Predefined possible values are reported in `Courier`. The default value is indicated in the table and underlined.

Note In the tables below we use the following convention for the type of variables:

C	character (a string delimited by blanks, tabs or apices)
R	real (a number in floating point or engineering notation)
I	integer (an integer number)

Typing must be respect in the input file to avoid errors or mis-interpretation by the parser.

Branch

The `Branch` block defines the characteristic of an electrical branch, i.e. an elementary component of the circuit to be analysed. The branch number must follow the keyword `Branch.`

Variable	Type	Units	Meaning

`Type`	C	(-)	Branch type. Possible values:
			`CurrentSupply`
			`Inductance`
			`Resistance`
			`VoltageSupply`

`Current`	R	(A)	Current in the branch.
`Inductance`	R	(-)	Vector of `Branches` elements containing the inductance matrix of this branch with all the others.
`Model`	C	(-)	If `Type` is `CurrentSupply` describes the current model. Possible values: `Constant` constant in time, equal to `Current`. `Window` equal to `Current` from time 0 to `Tau`, zero otherwise. `User` user defined through the function `UserBranchCurrent`. If `Type` is `VoltageSupply` describes the voltage model. Possible values: `Constant` constant in time, equal to `Voltage`. `Window` equal to `Voltage` from time 0 to `Tau`, zero otherwise.

			`User` user defined through the function `UserBranchVoltage`. If `Type` is `Resistance` describes the resistance model. Possible values: `Constant` constant in time, equal to `Resistance`. `User` user defined through the function `UserBranchResistance`. `External` the branch resistance is obtained from one of the other CryoSoft simulators, through explicit coupling at each time step. This coupling requires execution under the SuperMagnet environment, and leads to an error in case it is used in stand-alone mode. See the SuperMagnet manual for more details.
`Resistance`	R	(Ω)	Resistance in the branch.
`Tau`	R	(s)	Voltage or current time constant, used if `Model` is `Window`.
`Voltage`	R	(V)	Voltage in the branch.

Mesh

The `Mesh` block is used to define the connectivity matrix, specifying how branches are connected in series (or anti-series) to formed closed loops of currents (or meshes) The mesh number must follow the keyword `Mesh.`

Variable	Type	Units	Meaning
`Connectivity`	I	(-)	Vector of `Branches` elements containing the connection matrix of the mesh with all the branches; either 0, 1 or –1. The three values correspond to the following cases: (1) the branch is in the mesh and the current is in the same direction as the mesh current; (-1) the branch is in the mesh and the current is in the opposite direction as the mesh current; (0) the branch is not in the mesh.
`Current`	R	(A)	Initial current in the mesh.

Simulation

The *simulation* block describes the numerical parameters for time integration, logging and storage of results.

Variable	Type	Units	Meaning
`Branches`	I	(-)	Total number of branches.
`EndTime`	R	(s)	End time to be reached with the simulation.
`LogFile`	C	(-)	Log file name. This file contains the echo of the input and the log of the run, including error messages. If not given the default log file name is `power.log`.
`Meshes`	I	(-)	Total number of meshes.
`OutputStep`	R	(s)	Time step for storage of the results. The results are written to the output binary file every `OutputStep` seconds of simulation.
`Restart`			Flag triggering a restart. If this key is present in this block POWER reads the content of the specified `StorageFile` until the last stored time is found. The simulation begins then from this time. Storage of results continues on `StorageFile` (appended). All input will be ignored, except for `EndTime`, `LogFile`, `OutputStep` and `TimeStep`.
`StartTime`	R	(s)	Start time for the beginning of the simulation.
`StorageFile`	C	(-)	Binary storage file name. This file contains the results stored at the user's specified times, and is used for restarts or post-processing. If not given the default file name is `power.store`.
`TimeStep`	R	(s)	Time step for the time integration of the set of partial differential equations describing the evolution of the network. A POWER run takes place at a constant time step. The integration is performed with a full implicit (Euler-Backward), first order algorithm that insures best robustness properties
`Title`	C	(-)	Problem title.

Variables

The *variables* block is used to define user variables, with given name and type, stored internally and shared among routines and procedures. The value of these user-defined variables is accessible through a simple calling protocol in FORTRAN, which greatly simplifies the preparation and parameterization of External Routines. Variables can be seen as an extension of the standard input parameters, i.e. a facility for easy customization.

Variables are defined with the following syntax:

VariableType *VariableName* Value

where *VariableType* is one of the types defined in the table below, *VariableName* is the name assigned to the variable, and used later to retrieve its value, and Value is the value, of the appropriate type, assigned to the variable.

Note We report below a short form of the variables syntax. For further reference, and for explanations on how to access variables from customized External Routines, consult the Variables manual [2]

VariableType	Meaning
`Character`	*VariableName* is a string, whose Value is read as a text, delimited by apexes if the text contains a blank (not recommended)
`Integer`	*VariableName* is an integer, whose Value is read according to FORTAN READ conventions
`Real`	*VariableName* is a real, whose Value is read according to FORTAN READ conventions (floating point or scientific notation)

The variables defined in the *variables* block are accessed from the External Routines (and elsewhere in subroutines and functions linked at run time) through calls to the function **`getXVariable`**`(VariableName,Value)`, where **`X`** stands for the variable type (i.e. **`C`**, **`I`** or **`R`**) as described in [2].

CHAPTER 5

Post-processing Language Reference

Structure and syntax

The post-processing command file is read by the post-processor interpreter of POWERPOST. This parses and analyzes the syntax and the grammar of the various entries. In general the file contains a series of commands that are executed in sequence during a post-processing session.

The structure and content of the post-processing command file is similar to that of the input file already described in Chapter 4. In particular the following rules and conventions apply:

- the identifier of a variable and the corresponding value(s) can appear at any position on the line, they can carry on to several lines and must be separated by blanks or tabs;
- the interpretation is case insensitive;
- abbreviations of the keys are not allowed;
- a character ';' in any position of the command line indicates that the remainder of the line must be considered as a comment. If the ';' is the first character in a line, then the whole line is ignored.

Parsing of the input file is finished as soon as an end-of-file or the `stop` command are found. At this point the post-processor completes all pending print-outs and plots and closes the session. For sample input files see Chapter 3.

Commands reference

Post-processing commands In this section we report the list of the postprocessing commands and their meaning in alphabetical order. The keywords identifying commands and options are given in `Courier`. Parameters and values for the commands are given in *italic*.

Note The selection of the items to plot or to print is done identifying first the *target*, i.e. quantity to be plotted/printed, and then the *support*, i.e. the component over which the quantity is defined. Each support must be followed by its identification number, coherent with the input simulation file (e.g. Mesh `2` for the second mesh component defined in the input for the simulation with POWER).

`NewPage`

Force a new plot page to be generated

`OutputFile` *name*

Set the name of the file for printed output (generated with the command `Print`). The default file name for printed output is `powerpost.out`. The file name can be changed only before the first printed output is generated. The command is ignored if a printed output has already been generated on another file or on the default file.

`Plot` *target* $support_1$ $support_2$ … $support_n$

Generate n plot frames of *target* for the specified *support(s)* as a function of time or space according to the selection done (see the `Select` command).
Example: `plot current branch 1 branch 2`

`Plot` $target_1$ $support_1$ `vs` $target_2$ $support_2$

Plot $target_1$ of $support_1$ versus $target_2$ of $support_2$ at all times or space positions selected (see the `Select` command).
Example: `plot voltage branch 1 vs voltage branch 2`

`PostScriptFile` *name*

Set the name of the file containing Postscript® output. The default file name for printed output is `powerpost.ps`. The file name can be changed only before the first plot is generated. The command is ignored if a PostScript® output has already been generated on another file or on the default file.

`Print` $target_1$ $target_2$ … $target_n$ $support_1$ $support_2$ … $support_m$

Generate a table of n x m columns of the *target(s)* in the *support(s)* for every time or space coordinate selected (see the `Select` command). Note that several targets and supports can be printed simultaneously.
Example: `print current voltage branch 1 branch 2`

`Query` *query option*

List to standard output the input setting of *query option*, this can be one of the *BlockName* identifiers as for the input simulation file (`Branch`, `Meshs`, `Simulation`) or `All` to list the complete input set.

`Reset EndTime`

Reset the end time for plots and listings to the last simulation time stored in the binary storage file.

`Reset StartTime`

Reset the start time for plots and listings to the first simulation time stored in the binary storage file.

`Set Color on/off`

Switch among color coding and dashed-line coding (B/W) for curves plotted for different supports in the same plot frame, default is `off` (i.e. dashed-line coding).

`Set EndTime` *t*

Set the end time for plots and listings, default is the last time stored in the binary storage file.

`Set PlotsPerPage` *n*

Set the number of plots per page. The number n must be an integer equal to 1, 2, 3, 4 or 6, 6 being the default. Changing the number of plots per page will automatically generate the plots to a new page

`Set StartTime` *t*

Set the start time for plots and listings, default is the first time stored in the binary storage file.

`Stop`

Stop execution and close the session. An end-of-file during parsing of the command file results in the same effect.

`StorageFile` *name*

Set the name of the file containing the binary stored results from POWER. The default file name for printed output is `power.store`. Opening and reading of the binary storage file is automatic after parsing the first command. Therefore this command, if present, must be the first in the post-processing command file.

Supports and targets All plotting and print-out actions of the post-processor POWERPOST need the selection of a target to be plotted/printed and the relative support. A target is a variables or an auxiliary quantity computed in the simulation (e.g. current). A support is the component on which the quantity is defined (e.g. branch component number 2). Target and support must be selected from a valid combination (e.g. current of branch component number 2). In the following table we report the keys for the valid combinations of targets and supports. Any invalid selection or combination of target and support results in a syntax error during parsing.

Support	Target	Units	Meaning
`Branch`	`Current`	(A)	Current in the branch
	`CurrentDerivative`	(–)	Current derivative in the branch
	`Voltage`	(V)	Voltage in the branch
	`VoltageDerivative`	(–)	Voltage derivative in the branch
`Mesh`	`Current`	(A)	Current in the mesh
	`CurrentDerivative`	(–)	Current derivative in the mesh

CHAPTER 6

External Routines

Warning External Routines give unlimited access to the data structure used by the main program. Improper programming of External Routines can therefore corrupt operation and lead to evident or concealed malfunctions and generate manifest or hidden errors in the computed results. IN NO EVENT WILL CRYOSOFT BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY AUTHORISED OR UNAUTHORISED USE OF THIS FEATURE, even if advised of the possibility of such damages.

Linking external routines

The External Routines for POWER are FORTRAN functions packaged in a series of files contained in the directory:

```
~/CryoSoft/usr/power/code_x.x
```

(where `x.x` stands for the version you received) which you will have received with the standard installation. In order to customize the code you will need to write modified version of these files. We strongly suggest to create your own directory tree within the above directory, and to modify only copies of the External Routines in order to be able to safely retrieve the standard version at your wish. Once the modified routines are ready, you will need to compile them and link them to the standard part of the code, to produce a customized version of the executable of POWER. For this purpose you can use the standard makefile

```
~/CryoSoft/etc/power.make
```

that can be copied and modified. Once more we strongly suggest that you modify only a copy of the standard makefile. Refer to the installation guide [1] for more details on the use of the makefiles, compilation and link-editing of the program.

Calling protocol

The following sections describe the calling protocol for the External Routines. For clarity we have subdivided the description in sections that are either associated with the type of function or with the type of component involved. The convention followed for the definition of the FORTRAN type of variables is the same as described in Chapter 4.

The External Routines for POWER are defined as FORTRAN `functions`. The `function` returns a single `real` or `integer` value that must be computed by the user within the routine. All parameters passed to the `function` must be regarded as input parameters and cannot be modified.

Note FORTRAN unit numbers above 50 are reserved by the CryoSoft library for internal use, and should not be allocated for read/write operations. Any allocation or use of units above 50 can result in I/O errors or malfunctions.

Branch resistance

```
real function userBranchResistance (Branch, RInitial, Voltage,
                                    Current, Time)
```

Returns the resistance (Ω) of a branch. Called if `Model=user` for a branch of type `Resistance`.

Parameter	Type	Units	Meaning
`Branch`	I	(-)	Branch number
`RInitial`	R	(Ω)	Initial resistance, as from input
`Voltage`	R	(V)	Voltage in the branch
`Current`	R	(C)	Current in the branch
`Time`	R	(s)	Simulation time

Branch current

```
real function userBranchCurrent (Branch, IInitial, Voltage,
                                 PreviousCurrent, Time)
```

Returns the current (A) of a branch. Called if `Model=user` for a branch of type `CurrentSupply`.

Parameter	Type	Units	Meaning
`Branch`	I	(-)	Branch number
`IInitial`	R	(□)	Initial current, as from input
`Voltage`	R	(V)	Voltage in the branch
`PreviousCurrent`	R	(A)	Current in the branch at the previous time step
`Time`	R	(s)	Simulation time

Branch voltage

```
real function userBranchVoltage (Branch, VInitial, PreviousVoltage,
                                 Current, Time)
```

Returns the voltage (V) of a branch. Called if `Model=user` for a branch of type `VoltageSupply`.

Parameter	Type	Units	Meaning
`Branch`	I	(-)	Branch number
`VInitial`	R	(V)	Initial voltage, as from input
`PreviousVoltage`	R	(V)	Voltage in the branch at the previous time step
`Current`	R	(A)	Current in the branch
`Time`	R	(s)	Simulation time

CHAPTER 7

Troubleshooting and Errors

Error messages are reported to the output ASCII log file and to the standard output. The form of a typical error report is the following

```
ERROR in procedure <procedure name>: <error message>
called by <calling procure> at position <n>
called by <calling procure> at position <m>
......
```

where <*procedure name*> is the name of the routine where the error occurred and <*error message*> reports a short description of the error situation. This line is followed by the trace of the <*calling procedure*> up to the main program. In case of queries about error conditions, please take care to report error messages completely, including the calling trace.

Errors can be generated at four different levels in the code:

- input parsing and syntax errors;
- data consistency errors;
- runtime errors;
- internal consistency errors.

Input parsing errors

Input parsing and syntax errors are detected during the interpretation of the input file. They indicate that the variable naming, the command syntax or the type and number of numerical data in the input file are incorrect. Verify syntax in the input file in this case.

Data consistency errors

Data consistency errors are detected when input data are not coherent among themselves and would result in a model that cannot be analyzed. Typical cases are selection of incompatible options, or input data out-of-range. Verify the consistency of the input data in this case.

Runtime errors

Runtime errors are detected either when the solver enters a physical or numerical instability, or when the size of the problem exceeds the maximum allowed. Physical instabilities can be triggered by improper setting of physical conditions (e.g. initial conditions or boundary conditions), or excessive transient loads in case of non-linear network simulations (e.g. very large voltages or current transients). Verify input conditions in this case.

Numerical instabilities can be triggered by the use of very large time steps. In case of numerical instability, attempt at reducing the maximum time step.

The maximum size of the network that can be solved is determined by the requested memory allocation in the FORTRAN include file:

```
~/CryoSoft/src/power/code_x.x/includes/parameters.inc
```

where parameters are set statically. The parameters affecting memory allocation are the following, with the associated meaning:

Parameter	Meaning
`MaxBrc`	maximum number of branches
`MaxMsh`	maximum number of meshes

The version of the code you received can be modified by adjusting these parameters as desired. The code then needs to be compiled and link-edited as explained in the installation manual you received [1].

Warning Modifying the code dimensioning parameters requires understanding of the memory allocation for the system variables, and of the internal structure of the code. IN NO EVENT WILL CRYOSOFT BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY AUTHORISED OR UNAUTHORISED USE OF THIS FEATURE, even if advised of the possibility of such damages.

Internal consistency errors

Internal consistency errors indicate corruption of the internal data structure of the program. An internal consistency error cannot be generated using the standard program and reading data from input only. However, they can be detected in case that customized External Routines with improper data handling are used. They diagnose a severe fault within the code. If you are using External Routines, verify their consistency with the calling protocol. In case you are not using External Routines, report internal consistency errors to us.

CHAPTER 8

References

[1] CryoSoft Installation Manual, Version 8.1, 2016.

[2] CryoSoft Variables Manual, Version 1.0, 2016.