Manual del usuario

sceptresimusol

Simulación de redes no lineales y sistemas dinámicos

Diego Saravia

ORCID: 0000-0002-1868-0409

dsa@unsa.edu.ar

Proyecto Simusol

Escrito con asistencia de IA: Claude (Anthropic), modelos Fable 5 y Opus 5 (2026).

Borrador — noviembre de 2026

Un manual nuevo, escrito desde cero, del SCEPTRE moderno que vive en `sceptre2novender/nuevo`. No es la traducción del manual de Novender, que está en `docs/manual-es`.

Índice general

1. Presentación **1**
- 1.1. Quiénes . . . 1
- 1.2. De dónde viene, y con qué convive . . . 1

2. Arranque rápido **3**
- 2.1. Qué entra . . . 3
- 2.2. Las opciones de sc1 . . . 3
- 2.3. Un consejo para los circuitos grandes . . . 4

3. El flujo de datos **5**
- 3.1. La entrada . . . 5
- 3.2. Lo intermedio: lo que la fase 1 le deja a la fase 2 . . . 5
- 3.3. La salida . . . 5
  - 3.3.1. Los cuatro formatos de resultados . . . 6
- 3.4. El flujo puro: stdin, stdout, stderr y el código . . . 6
- 3.5. El papel de sc1 . . . 7
- 3.6. El flujo, de un vistazo . . . 7
- 3.7. Los lotes . . . 7
- 3.8. El servidor . . . 7
- 3.9. Transportar los pedidos de dibujo . . . 7

4. El lenguaje del deck **9**
- 4.1. Modelo frente a datos y usos . . . 9
  - 4.1.1. La política de la separación . . . 9
- 4.2. Las funciones: los tres motores (capítulo 5) . . . 10
- 4.3. El lector del deck: slex . . . 10
- 4.4. Funciones de usuario en Fortran o en C . . . 10
  - 4.4.1. Las funciones reconstruidas por ingeniería inversa . . . 11
- 4.5. La capacidad total . . . 11
  - 4.5.1. CALCULA y EXPRTK desde el deck . . . 11

5. Los tres motores **13**
- 5.1. tabular2: las tablas . . . 13
- 5.2. ExprTk: expresiones explícitas . . . 13
- 5.3. calcula: sistemas implícitos . . . 13
- 5.4. Versiones mínimas, y lo que el protocolo todavía no lleva . . . 13

6. La corrida: dinámico, reentrante, sin topes **15**
- 6.1. El circuito como `.so` . . . 15
- 6.2. Reentrancia e instancias . . . 15
- 6.3. El servidor y los lotes . . . 15
- 6.4. El latido . . . 15
- 6.5. Memoria dinámica . . . 16

7. El álgebra: LAPACK, BLAS y UMFPACK **17**
7.1. La matriz en datos: `SCEPTRE_MATRIZ=numerica` . . . 17
7.1.1. Por qué crecía, y qué se volcó . . . 17
7.2. Lo medido, y dónde paga cada cosa . . . 18

8. Las llaves del proceso **19**

9. Diagnósticos **20**
9.1. El renglón de salida y los códigos . . . 20
9.2. El origen de cada diagnóstico . . . 20
9.3. El mapa del programa generado . . . 20
9.4. Los avisos de recursos . . . 21

10. Verificación: el corpus **22**

11. Construcción e instalación **23**
11.1. Autotools: configure, automake y libtool . . . 23
11.2. La instalación a través del tiempo: el caso de Dia . . . 23
11.3. El MANIFIESTO . . . 24

12. Cómo pedirle los comportamientos de antes **25**

13. Agregados documentados durante la traducción **26**
13.1. Capacidad total y capacidad diferencial . . . 26
13.2. Restricciones y reconocimiento del modificador TOTAL . . . 27
13.3. Reparaciones de la convolución . . . 27
13.3.1. Cómo se revivió, y por qué pide paso fijo . . . 28
13.4. Corrección de lectura de los núcleos de convolución . . . 28
13.5. Dimensionamiento del ejemplo de convolución reparado . . . 29
13.6. Corrección de la desactivación de la compresión . . . 29
13.7. Separación interna de la selección vectorial . . . 29
13.8. Fin de archivo en la lectura del monitor de terminal . . . 29
13.9. Capacidad histórica y ampliaciones de la implementación actual . . . 30
13.10. Validación de TIME, FREQ y parámetros complejos . . . 30
13.11. Construcción con Autoconf, Automake y Libtool . . . 31
13.12. Empaquetado de SCEPTRE mediante SummaPack . . . 32
13.13. Tabular: lectura de resultados y conversión de tablas . . . 33
13.14. Las tres generaciones del formato binario Dirac . . . 33
13.15. Plan de modernización: objetivos, resultados y avance . . . 34
13.15.1. Estado de los pasos del plan . . . 35
13.15.2. Hitos y ampliaciones posteriores . . . 36
13.15.3. Qué permite afirmar el avance registrado . . . 36
13.15.4. Pendientes que siguen teniendo entidad propia . . . 36
13.16. Reconstrucción de funciones auxiliares de 32 bits . . . 37
13.17. Alcance de TOTAL frente a las inductancias . . . 37
13.18. Archivos .d del corpus que incluyen extensiones de Siru . . . 37
13.19. Mapa de la ampliación documental con el corpus general . . . 38
13.19.1. Criterios para ampliar los casos . . . 39
13.20. COMPUTER TIME LIMIT . . . 40
13.21. El paquete binario y el cambio de sistema operativo . . . 41

13.21.1 Lo que el paquete necesita en cada corrida . . . 41
13.21.2 Por qué un paquete binario muere al cambiar el sistema . . . 41
13.21.3 La capacidad Ajolote: reconstruirse desde las fuentes que trae el paquete 41
13.21.4 El paralelo con DKMS . . . 42
13.22 Los archivos propios del usuario: `sceptre-user-setup` . . . 42
13.22.1 Qué hace, en orden . . . 43
13.22.2 Qué instala el paquete, y dónde . . . 43
13.22.3 Para qué sirve `modellib.dat` . . . 44
13.22.4 Quién manda sobre quién . . . 44
13.22.5 Qué reemplaza . . . 44
13.23 Las funciones propias del usuario, como módulo dinámico . . . 45
13.23.1 El lugar . . . 45
13.23.2 El método . . . 45
13.23.3 Cómo se escribe una función . . . 45
13.23.4 Cómo se la llama desde el deck . . . 46
13.23.5 Y si está escrita en C . . . 46
13.23.6 Dos cuidados que el mecanismo se toma . . . 46
13.23.7 Lo que lo comprueba . . . 47
13.23.8 Qué pasó con `usrlib.a` . . . 47
13.23.9 Ejemplo desde Dia . . . 47
13.24 Los modelos propios del usuario, y cómo entran . . . 47
13.24.1 El lugar . . . 47
13.24.2 Cómo se escribe un modelo . . . 48
13.24.3 Cómo se usa desde el circuito . . . 48
13.24.4 Quién gana . . . 49
13.24.5 Qué pasa al pasarlos . . . 49
13.24.6 Lo que queda en el listado . . . 49
13.24.7 Lo que lo comprueba . . . 50
13.24.8 La fase 1 ya no abre ninguna biblioteca . . . 50
13.24.9 Editar la biblioteca desde un deck: suspendido . . . 50
13.24.10 Cómo se leía, para el que quiera seguirlo . . . 51
13.25 La lectura por slex y el `.dsa` directo . . . 51
13.25.1 Los tres tratamientos de la entrada, hoy . . . 51
13.25.2 El `.d` por tokens: dónde quedó cada análisis . . . 52
13.25.3 La política de la entrada . . . 53
13.25.4 El lector de dsa: libdsa . . . 54
13.25.5 El camino al `.dsa` directo, y sus varas . . . 54
13.26 El lector por tokens del `.d`, por dentro . . . 55
13.26.1 Cómo entra slex . . . 55
13.26.2 Las reglas . . . 56
13.26.3 Los lectores por sección, con sus errores . . . 56
13.26.4 El valor por gramática (`valor.y`) . . . 56
13.26.5 Las varas . . . 57
13.27 Los modelos expandidos en C, y las tablas en tabular2 . . . 57
13.28 Lo que el modelo declara para los usos: los tres programas . . . 58
13.29 La reentrancia: varios circuitos en un mismo proceso . . . 58
13.29.1 La instancia, por intercambio . . . 58
13.29.2 La fase 1 reentrante . . . 59
13.29.3 La fase 2 reentrante, hasta alternar paso a paso . . . 59

13.30 El circuito como objeto dinámico, y el servidor . . . 59
13.31 La memoria dinámica: cómo desaparecieron los topes . . . 60
13.32 La modernización del Fortran, y por qué hacía falta . . . 61
13.32.1 Las construcciones borradas . . . 61
13.32.2 El susto de `-O2`, que era un síntoma . . . 61
13.32.3 Los COMMON, bloque por bloque . . . 61
13.32.4 Los `ENTRY` y los `GOTO` . . . 61
13.32.5 El Fortran que el SCEPTRE *escribe* . . . 62
13.33 Las clases del SCEPTRE, y qué prueba el corpus . . . 62
13.33.1 Qué son las clases . . . 62
13.33.2 Qué parte del código no prueba el corpus . . . 62
13.34 El plan de modernización, paso por paso . . . 63
13.34.1 Dos cosas que el plan anotó como límites y dejaron de serlo . . . 63

A. Códigos de salida **65**

B. Ejemplos **68**
B.1. La descarga que resuelve un motor: calculaohm y exprtkohm . . . 68

1 Presentación

sceptresimusol es un simulador de redes eléctricas no lineales y sistemas dinámicos. Desciende de SCEPTRE (IBM, 1966) por la línea que mantuvo Wolf-Rainer Novender, pero este manual no describe aquel programa: describe uno nuevo, reescrito en Fortran moderno, que conserva el lenguaje de entrada `.d` –modificado donde hizo falta– y cambia casi todo lo demás. La traducción del manual histórico está aparte, en `docs/manual-es`; este libro se escribe desde cero y presenta el comportamiento nuevo como el normal. El capítulo 12, al final, explica cómo pedirle los comportamientos de antes.

Lo nuevo, en una página:

- **Modo dinámico**: el circuito se compila una vez como biblioteca compartida (`.so`) y se reutiliza entre corridas (capítulo 6).
- **Reentrante**: varias instancias del mismo circuito, o circuitos distintos, conviven en un mismo proceso; el estado vive en instancias, no en variables estáticas.
- **Memoria dinámica**: las tablas crecen con el problema; los topes fijos del SCEPTRE histórico desaparecen.
- **Álgebra por bibliotecas**: LAPACK/BLAS para lo denso y UMFPACK para lo disperso (capítulo 7); la matriz del sistema puede armarse numéricamente en vez de generarse como texto.
- **Separación modelo / datos y usos**: el modelo es una cosa y los datos con que se corre son otra, también en los archivos (`M` y `DU`, capítulo 4).
- **Tres motores llamables desde el deck**: tabular2 para las tablas, ExprTk para expresiones explícitas y calcula para sistemas implícitos (capítulo 4).
- **Llaves de proceso**: el comportamiento se gobierna con variables `SCEPTRE_*` y opciones de `sc1` (capítulo 8), sin recompilar nada.

1.1. Quiénes

El desarrollo inicial de sceptresimusol fue de **Lorena García**; hoy lo administra **Diego Saravia**, dentro del Proyecto Simusol.

1.2. De dónde viene, y con qué convive

IBM desarrolló SCEPTRE en 1966 por encargo del gobierno de los Estados Unidos: análisis automático de redes no lineales, en Fortran, con el transitorio, el punto de operación y los parámetros definidos con sus derivadas como rasgo distintivo. La revisión **SC1** es casi idéntica a la original; **Wolf-Rainer Novender** (THM Friedberg) mantuvo esa línea —SC1 con el posprocesador SD2— durante décadas, y su *Benutzerhandbuch* es el manual histórico, traducido en `docs/manual-es`. Este árbol parte del último fuente de Novender (`sc1-sd2.26.131`, mayo de 2026): la copia `cuasioriginal` —el tarball más lo mínimo para

compilar limpio— queda como referencia intocable, y sobre su copia se hizo todo lo que este manual cuenta.

Simusol, el proyecto que da nombre al paquete, simula sistemas —los solares en su origen— dibujándolos con Dia y resolviéndolos con SCEPTRE; sceptresimusol nació como el SCEPTRE de Simusol. **siru** es el simulador nuevo del mismo proyecto, y los dos comparten las piezas: el formato `.dsa` (`siru/AcuerdosGenerales_dsa.md`) como lengua común de modelos, datos y usos; **slex** como único lexer y lector de los formatos (capítulo 4 y el agregado 13.25); y el corpus de circuitos con que ambos se verifican. Dentro de ese ecosistema, sceptresimusol es el motor SCEPTRE: siru corre los mismos modelos con sus propios motores (siru, Calcula, mtk) y con éste, y los resultados se comparan entre sí.

2 Arranque rápido

Un caso se corre con el lanzador `sc1`:

```
sc1 CASO.d
```

`sc1` encadena las dos fases: la fase 1 lee el deck y genera el programa del circuito; la fase 2 lo carga como `.so` (el modo dinámico, que es el normal) y simula. Las salidas quedan junto al caso: `CASO.log` (el listado de las dos fases, en *append*: acumula corridas y la última es la que vale), `CASO.dirac` (la base de resultados), `CASO.sim` (los resultados en texto) y los `datasave`/`acsave` que la corrida deje. El código de salida del proceso resume qué pasó (capítulo 9).

2.1. Qué entra

`CASO.d` el deck SCEPTRE, tal cual.

`CASO.ds` / `CASO.ds2` un circuito de siru: se traduce a `.d` con `d2datdinc` y se corre el `.d`.

`CASO.dsa` el formato declarativo: la fase 1 lo lee directo a sus variables, sin traducirlo a `.d`. `sc1 X.dsa` es el archivo reunido; `sc1 X.M.dsa` es el modelo solo, y busca sus datos y usos en `X.DU.dsa`, al lado (capítulo 4). `sc1` separa antes el `PARAPLOT`, que es del lanzador, a `CASO.graficos.ds`.

`-` el modo filtro: el deck entra por la entrada estándar, el `.sim` sale por la salida estándar y todo lo demás va a la de error (capítulo 3).

2.2. Las opciones de sc1

`-o DIR` directorio que recibe las salidas.

`-m ARCHIVO` la biblioteca de modelos en lugar de la instalada (`modellib.dat`); los modelos propios se suman aparte, véase 13.24.

`--sin-modelos` sin los modelos propios: la instalada sola.

`-u ARCHIVO` biblioteca de funciones de usuario propia (compilada con `-fPIC`: se enlaza adentro del `.so`).

`-nombre NOMBRE` cómo se llama la corrida: nombra el `.log`, el `.sim` y todo lo demás.

`-continuar` conserva lo de la corrida anterior, para los decks `CONTINUE` y `RE-OUTPUT`.

`-sodir DIR`, `-rehacer` dónde queda el par `.so`+huella y cuándo rehacerlo (capítulo 6).

`-sinlatido` sin tiempo de corte, para la corrida que se sabe larga (capítulo 6).

`-lote CASO1.d CASO2.d …` varios decks con un solo `sceptreDU` servidor (capítulo 6).

`--parar`, `--verbose`, `--lector`, `--tablas`, `--modelos`, `-estatico` las llaves con forma de opción (capítulos 8 y 12).

2.3. Un consejo para los circuitos grandes

Los temporales de la fase 1 van a `TMPDIR` (o `/tmp`), que en las máquinas nuevas vive en RAM y con cuota. Para un circuito grande conviene siempre:

```
mkdir -p ~/tmp
TMPDIR=~/tmp sc1 CASO.d
```

Si el lugar se acaba, la fase 1 avisa en el listado (capítulo 9).

3 El flujo de datos

De la entrada a la salida, una corrida es una cadena de archivos que `sc1` encadena. Conocerla explica qué es cada cosa que queda en el directorio.

3.1. La entrada

El caso entra como `CASO.d` (el deck SCEPTRE; un `.ds` se traduce antes) o como `CASO.dsa`, que no se traduce: la fase 1 lo reconoce por su primer renglón y lo carga directo a sus variables. `sc1` prepara `input.dat` –la entrada tal como la lee la fase 1– y guarda a su lado `input.dat.origen` con el nombre del archivo original, para que todo diagnóstico pueda nombrar al verdadero culpable.

3.2. Lo intermedio: lo que la fase 1 le deja a la fase 2

La división del deck en modelo y datos/usos (capítulo 4) es también la división del trabajo entre las fases: **la M es de la fase 1** –el modelo es lo que se valida, se genera y se compila al `.so`– y **la DU es de la fase 2** –los datos y los usos alimentan cada corrida sin tocar el modelo compilado–. Cambiar datos no obliga a regenerar el circuito; cambiar el modelo, sí.

Y la división la administra **sc1, no el programa sceptre**: los archivos `M` y `DU` son cosa del lanzador, que los junta y prepara la entrada única que cada fase ve. `sceptreM` y `sceptreDU` no saben de `M.d` ni de `DU.d`: ven una entrada y producen una salida, como un filtro. Eso mantiene al simulador simple y deja la política de archivos –qué se junta con qué, qué se reutiliza– en un guion que cualquiera puede leer y cambiar.

La fase 1 (`sceptreM`) lee `input.dat`, valida el deck y deja:

`pgmsave.for` el programa generado: el Fortran del circuito (las ecuaciones, `DIFFUN`, las salidas).

`datasave.dat` los datos numéricos del caso: valores, condiciones iniciales, controles de corrida.

`matsave.dat` con `SCEPTRE_MATRIZ=numerica`, los términos ralos de la matriz del sistema (capítulo 7).

`acsave.dat`, `plotsave.dat` lo propio de AC y de los datos de ploteo, cuando el deck los pide.

`sc1` compila entonces el generado: en el modo dinámico, a `CASO.so` (con huella, reutilizable); en el estático, a un `sceptreDU` propio del caso.

3.3. La salida

La fase 2 carga el circuito, lee `datasave.dat` y simula. Deja:

`CASO.log` el listado único de las dos fases (`SCEPTRE_LOG`), abierto en *append*: acumula corridas; la última es la que vale.

`CASO.dirac` los resultados tabulados de todas las corridas; `interpola --exportar-simul` los traduce a `CASO.sim`.

`CASO.datasave-final` el estado final, el que esperan `CONTINUE` y `RE-OUTPUT` de una corrida de continuación.

el código de salida del proceso, estable por causa (capítulo 9).

3.3.1. Los cuatro formatos de resultados

Los resultados existen en *cuatro* formas: tres generaciones del binario Dirac y la salida de texto.

1. **POS antiguo** (binario, heredado de 00.317–2013): firma `DIRECTOR`, arreglos separados de 51 nombres en el registro inicial.
2. **POS4** (binario, el del árbol SC1/SD2 de partida): cabecera `DIRECTORY`, directorio en un único registro de 2048 bytes –por eso el tope histórico de 63 vectores.
3. **POS4 extensible** (binario, *el actual*, ampliación del proyecto): la cabecera dice cuántos registros ocupa el directorio, así que entran todas las salidas; con hasta 63 vectores el archivo es byte a byte el POS4, y leer uno más grande pide tabular2 2.0-25 o posterior.
4. `CASO.sim` (texto): los resultados tabulados de todas las corridas, lo que usa Simusol. Se escribe *antes* que la base binaria: si la base falla, el `.sim` ya está completo.

La fase 2 escribe el binario por `fort.7` y `sc1` lo renombra a `CASO.dirac`; el detalle byte a byte de las tres generaciones está en el agregado 13.14.

3.4. El flujo puro: stdin, stdout, stderr y el código

Además de los archivos, la corrida respeta la disciplina Unix de los tres canales y el código de salida:

entrada estándar las fases *no* leen de stdin: `sc1` se las cierra con `</dev/null`, para que ningún «press ENTER» heredado se coma una tubería. Cuando `sc1` actúa de filtro, el deck entra por stdin y se copia a un temporal antes de correr.

salida estándar es del *resultado*. En el modo filtro, el `.sim` sale por el stdout verdadero y todo lo demás se desvía a stderr; en el uso normal, stdout de `sc1` lleva el progreso legible y nada que un programa deba parsear.

salida de error es de los *diagnósticos*: los renglones fijos `SCEPTRE: SALIDA | código | hora | archivo:línea | rutina | mensaje`, `SCEPTRE: ORIGEN | archivo | mensaje` y los complementos (mapa, matriz singular) van a stderr, además del listado. El protocolo máquina de calcula sigue la misma regla: su stdout es del protocolo y sus diagnósticos van a stderr.

código de salida es la palabra final y es *estable por causa*: un llamador decide por el número sin leer texto (apéndice A). Cero es éxito; los números se asignan por debajo de 126 porque de ahí para arriba el shell los confunde con los suyos (126 «no se puede ejecutar», 127 «no se encuentra», 128+N «murió por la señal N»).

3.5. El papel de sc1

`sc1` es el guion que sabe el orden: preparar la entrada, correr la fase 1, compilar o reutilizar el `.so`, correr la fase 2, repartir los archivos con el nombre del caso y traducir los resultados. Las fases pueden correrse a mano (`sceptreM`, `sceptreDU`), y el corpus lo hace, pero el camino normal es `sc1`.

3.6. El flujo, de un vistazo

```
CASO.d | CASO.ds(2) | CASO.M.dsa+CASO.DU.dsa
   |         (el .ds se traduce; el .dsa entra tal cual)
   v
  sc1 ----> input.dat ----> sceptreM (fase 1)
                              |-- pgmsave.for --> gfortran --> CASO.so
                              |-- datasave.dat (+ acsave, matsave)
                              v
                            sceptreDU (fase 2, SCEPTRE_SO=CASO.so)
                              |-- CASO.log (el listado)
                              |-- sceptre.sim / CASO.simnativo (la tabla)
                              |-- fort.7 --> CASO.dirac (--> interpola)
```

3.7. Los lotes

`sc1 -lote CASO1.d CASO2.dsa …` corre varios decks con *un solo* `sceptreDU` que queda corriendo (`sceptreDU --servidor`, dos FIFO: pedidos y respuestas). La fase 1 de todos los decks que la necesitan la hace un único `sceptreM --lote`, cada deck en un hijo y con su nombre adelante de cada renglón; después cada deck sigue como una corrida común —el `.so` con su caché por huella, sus archivos— y la fase 2 se le pide al servidor en vez de levantar un proceso por deck. Al final sale un renglón por deck y el código es el primero que no sea cero.

3.8. El servidor

`sceptreDU --servidor` lee pedidos por el descriptor 8 y contesta por el 9: cada pedido es un directorio de corrida con su `.so`, sus huellas y sus llaves, y la respuesta es el código de esa fase 2. Es la variante del flujo que evita pagar el arranque del proceso por cada corrida; `sc1` lo usa en `-lote` (con `SC1_SERVIDOR=1` en cada hijo) y `tests/prueba-servidor.sh` lo ejercita solo.

3.9. Transportar los pedidos de dibujo

`PARAPLOT` acompaña al caso, pero no entra en las ecuaciones. El lanzador separa sus órdenes antes de la fase 1 mediante `d2datdinc --separar-paraplot`; conserva el texto

restante del deck. Para un par DSA, recupera los bloques del DU. Al finalizar reúne los pedidos y la tabla en `CASO.graficos.ds`. La tabla se obtiene del POS4 mediante el exportador existente `sceptre2simul`, que conserva nombres de columnas y componentes complejas. En modo filtro (`sc1 -`), el documento sale por stdout y los mensajes por stderr. Sin PARAPLOT se conserva la salida numérica habitual. Las órdenes gráficas se ejecutan únicamente al llegar a Simplot; Sceptre no las evalúa. Las fuentes externas nombradas deben estar disponibles al graficar.

4 El lenguaje del deck

El lenguaje de entrada sigue siendo el `.d` de SCEPTRE –secciones `CIRCUIT DESCRIPTION`, `ELEMENTS`, `DEFINED PARAMETERS`, `FUNCTIONS`, `OUTPUTS`, `RUN CONTROLS`, `MODEL DESCRIPTION`– con modificaciones al servicio de lo nuevo.

4.1. Modelo frente a datos y usos

Un modelo es una cosa; los datos con que se corre y lo que se le pide son otra. El deck puede dividirse en dos archivos: `CASO.M.d` (la `MODEL DESCRIPTION`) y `CASO.DU.d` (el circuito que lo usa, sus datos y sus controles); concatenados forman el deck completo. La misma división existe en el formato `.dsa` (`M.dsa` y `DU.dsa`). La división no es cosmética: la M corresponde a la fase 1 (lo que se compila al `.so`, reutilizable) y la DU a la fase 2 (lo que cambia entre corridas); véase el capítulo 3.

4.1.1. La política de la separación

La frase que la define (27/10/2026): *la primera etapa genera la parte matemática del modelo; la segunda lo usa con datos y usos de cada corrida.* El reparto:

MODELO ecuaciones, elementos, conexiones, estados, parámetros declarados, funciones, tablas y recursos que definen el sistema. Es lo que la fase 1 valida, genera y compila al `.so`.

DATOS los valores de cada caso: parámetros, condiciones iniciales, lo que el modelo permite suministrar desde afuera.

USOS qué se le pide en esa corrida: punto de operación, transitorio, AC, sensibilidad, optimización, peor caso, Monte Carlo, salidas, continuaciones.

La prueba decisiva: preparar un modelo *una* vez, cerrar la fase 1, y correr con ese mismo módulo varias combinaciones de datos y usos sin recompilar. La medición que ordenó el trabajo (28/10/2026, la fase 1 sobre variantes de un mismo deck) dijo cuatro cosas:

1. **Los datos ya están separados**: cambiar un valor cambia un número de `datasave.dat` y nada del código generado.

2. **Las salidas y los controles son baratos**: media docena de renglones (`LOUTR`) y entradas de `cntrl`.

3. **Sensibilidad, optimización, peor caso y Monte Carlo son un solo bloque, el adjunto**: lo caro no es cuál de los cuatro se pide sino que haya alguno (de 499 a 665 renglones del generado).

4. **AC es otro modelo, no otro uso**: la linealización y las fuentes AC generan sus propias rutinas.

De ahí la decisión (28/10/2026, también en la especificación del `.dsa`): el MODELO declara `OBSERVABLES`, `VARIABLES` y `LINEALIZADO` —capacidades—, y el DU elige entre ellas. Y la política de caché: cambiar un valor, un conjunto, un objetivo, las frecuencias o las salidas *no recompila*; cambiar ecuaciones, tipos de elementos o conexiones sí, porque cambia el modelo. La huella del módulo incluye el modelo, sus recursos y las versiones del generador y el compilador.

Lo que ya corre de esta política: la carga de datos por nombre sin fase 1 (`SCEPTRE_DATOS`, un archivo `NOMBRE = VALOR` o un `.DU.dsa` directo), el `.so` reutilizado por huella, el servidor de la fase 2, y las tres palabras declarativas del modelo. Las etapas que faltan —el descriptor del modelo, los usos preparados en la fase 2 (el adjunto construido allá), AC desacoplado— están en `pasos/PENDIENTES`.

4.2. Las funciones: los tres motores (capítulo 5)

En `FUNCTIONS` conviven tres maneras de calcular:

`TABLE` **(tabular2)** las tablas del deck se interpolan con la biblioteca tabular2, con la semántica documentada del interpolador.

`EXPRTK` una expresión ya despejada se evalúa en el propio proceso con ExprTk:

```
FI = EXPRTK ('v/r', V=VJD, R=PRD)
```

`CALCULA` un sistema implícito se resuelve llamando al programa `calcula` (protocolo máquina); el primer argumento es el módulo, el segundo la variable que esa F devuelve, y los pares `dato=expresión` las entradas. Las F que comparten módulo y entradas hacen un solo `EVAL` que trae todas las salidas:

```
FI = CALCULA ('calculaohm.din', I, V=VJD, R=PRD)
FP = CALCULA ('calculaohm.din', P, V=VJD, R=PRD)
```

El módulo puede ser un archivo del modulodinamico, un sistema compilado de calcula, o un modelo definido en `.M.dsa`.

4.3. El lector del deck: slex

Desde octubre de 2026 el deck se lee por omisión con **slex**, el lexer y las gramáticas compartidas del proyecto, y el lector histórico queda a elección: `sc1 --lector propio|slex`. Con slex el deck va entero por tokens –una sola tokenización con reglas declaradas, en vez del rebanado de columnas de siempre– y los lectores por sección (elementos, parámetros, funciones, salidas, controles) se despachan sobre esos tokens; los dos lectores llenan las mismas tablas y los volcados prueban que dan lo mismo. Sobre slex se apoyan también `d2datdinc` –el traductor que convierte `.ds`/`.ds2` a `.d` y el que expande los `MODEL` fuera de la fase 1 con `--modelos c`– y el lector del `.dsa`, que arma el dato entero en memoria y la fase 1 recorre sin analizar texto: el mismo lector para todos los formatos de entrada, y ninguna traducción intermedia para el `.dsa`.

4.4. Funciones de usuario en Fortran o en C

Una función que el deck nombra en una expresión –`R1, 1-2 = X1 (MIA(TIME, P1))`– puede escribirse afuera y enlazarse:

1. **En Fortran**: una `FUNCTION` de `real*8` con argumentos `real*8` (el patrón está en `extras/`: `cuelga.f90`, `fmos.f90`). Cómo se escribe una, con sus reglas y un ejemplo completo, está en 13.23.
2. **En C**: se escribe la función en C y se la llama desde un envoltorio Fortran con `iso_c_binding` (`bind(c)`), que es el patrón de las extras históricas (`STIME`/`geometry`).
3. **No hay que compilar nada a mano**: el `.f90` se pone en `~/.config/sceptre/funciones/` y `sc1` arma solo la biblioteca dinámica y la enlaza, en los dos modos (13.23). El `.so` entra en `CASO.huella`, así que cambiar una función rehace el circuito. Quien ya tenga su propia biblioteca *estática* la sigue pasando con `-u`, y ahí el `-fPIC` importa: se enlaza *adentro* del `.so` del circuito.

4.4.1. Las funciones reconstruidas por ingeniería inversa

La `usrlib.a` heredada —que ya no existe: sus funciones son fuentes del proyecto y están en `extras/`, compiladas con la fase 2— traía dos objetos de 32 bits sin fuente en ningún árbol (`i01.o` y `c10.o`): todo deck que usara `FMOS`, o `FTABL1`/`FALL`/`FIZK`/`FTOR` y los moduladores, moría en el enlazador. Se reconstruyeron por *ingeniería inversa* del desensamblado x87, con los decks que las llaman como diccionario de los argumentos, cuidando el orden de las operaciones para reproducir el redondeo: de `i01.o` salió `FMOS` (un Shichman–Hodges cuadrático con drenaje y fuente intercambiables, exactamente lo que haría un ejemplo de MOS de 1970) y de `c10.o` las ocho de `c10.f90` (`FTABL1`, `FALL`, `FIZK`, `FTOR`, `FFREI`, `FRECHT`, `FSYNC`, `FTRAP`). La mejor verificación: siru, con otro integrador y la misma `FMOS`, da igual que SCEPTRE en el ejemplo `i01` (peor diferencia 0.00049). El método completo, paso a paso, está en `extras/README.md`; `extras/prueba.f90` recorre cada rama de cada función con el resultado a mano al lado, y `extras/objetos/LEEME.md` dice qué se sabe de los cuatro objetos que todavía no tienen fuente.

Hay además funciones *nuevas*, que no estaban: `CUELGA(t,t0)` (`cuelga.f90`), que se cuelga a propósito para probar el latido (la usa `corpus/cuelga.d`, que tiene que salir con 118 guardando los puntos).

Si el deck declara una función que no está en ninguna biblioteca, el error lo da el enlazador y queda en `CASO.compila`, no en el listado de SCEPTRE. Una función de usuario mal escrita que no vuelve no cuelga la corrida para siempre: la corta el latido (capítulo 6).

4.5. La capacidad total

SCEPTRE toma la capacidad como diferencial: $i = C(v)\,dv/dt$. Este árbol entiende además la *total*, $Q = C(v)\,v$, $i = dQ/dt$, con la palabra `TOTAL` al principio del valor:

```
C1, 2-0 = TOTAL Q1(VC1)
```

Sólo en un capacitor, y el valor sólo puede depender de la tensión de ese capacitor; `C1` en `OUTPUTS` da dQ/dv, la capacidad que se integra. El detalle y las restricciones están en los agregados 13.1 y 13.2.

4.5.1. CALCULA y EXPRTK desde el deck

La forma acordada (03/11/2026) para nombrar los motores en `FUNCTIONS`:

```
FUNCTIONS
  FI = CALCULA ('calculaohm.din', I, V=VJD, R=PRD)
  FP = CALCULA ('calculaohm.din', P, V=VJD, R=PRD)
  FE = EXPRTK ('v/r', V=VJD, R=PRD)
```

Cada `F` sigue siendo escalar, como exige SCEPTRE. En `CALCULA`, el primer argumento es el módulo (un archivo del modulodinámico, o un sistema compilado si el nombre no tiene `/` ni punto), el segundo la variable que esa `F` devuelve, y los pares `dato=expresión` las entradas; las `F` que comparten módulo y entradas se juntan por debajo en *un* solo `EVAL` que pide todas las salidas. En `EXPRTK` el primer argumento es la expresión, que se compila una vez y queda cacheada por su texto y sus nombres.

El brazo de la fase 2 está hecho y probado: `common_calcula_2.f90` (una sesión por módulo, con su proceso `calcula --machine`, su tubería y su caché del último punto; protocolo de calcula 6.0, `READY`/`EVAL`/`OK`; toda falla corta con el código 92) y `common_exprtk_2.f90` con su puente C++ (`exprtk_puente.cpp`, el único C++ del árbol; un error de compilación de la expresión corta con el 93). **Lo que falta es la fase 1**: reconocer `CALCULA(…)` y `EXPRTK(…)` en `FUNCTIONS` y generar las llamadas; los decks de ejemplo (`calculaohm`, `exprtkohm`, apéndice B) esperan en el corpus sin patrón hasta entonces (`pasos/PENDIENTES`).

5 Los tres motores

Tres bibliotecas externas hacen cuentas por el simulador. Las tres se llaman desde el Fortran de la fase 2; el deck las nombra en `FUNCTIONS` (capítulo 4).

5.1. tabular2: las tablas

tabular2 es la biblioteca de tablas e interpolación del proyecto. Las `TABLE` del deck se interpolan con ella por omisión (`ETABLE`); su semántica –orden, extrapolación, derivadas– está documentada en la propia biblioteca, y el interpolador histórico de SCEPTRE queda disponible con `--tablas sceptre`. También traduce los resultados (`interpola --exportar-simul`).

5.2. ExprTk: expresiones explícitas

ExprTk es un evaluador de expresiones en C++ (`exprtk.hpp`). La función `EXPRTK` del deck le pasa una expresión ya despejada –`'v/r'`– y sus variables; cada expresión distinta se compila una sola vez y queda cacheada. Corre en el propio proceso: no hay otro programa de por medio. Es el camino barato cuando la ecuación tiene la variable despejada.

5.3. calcula: sistemas implícitos

calcula es el resolvedor de sistemas algebraicos del proyecto (ancestro: psicro). La función `CALCULA` del deck lanza un proceso `calcula --machine` por módulo y le pide puntos por el protocolo de líneas (`EVAL`): se imponen los datos, Newton resuelve el sistema y vuelven las salidas pedidas. Las F que comparten módulo y entradas se juntan en un solo `EVAL`. El módulo puede ser un archivo del modulodinamico (`.din`), un sistema compilado, o un modelo en `.M.dsa`. Es el camino cuando la ecuación –o el sistema– no está despejado.

5.4. Versiones mínimas, y lo que el protocolo todavía no lleva

El motor calcula tiene que hablar el protocolo máquina de **calcula 6.0** (versión 1: saludo `READY 1`, pedido `EVAL`, respuesta `OK`/`ERR`); para definir el modelo en `.dsa` hace falta además el comando `DATOS` del protocolo, que las versiones viejas instaladas (r334) no tienen. El ejecutable lo elige `SCEPTRE_CALCULA`, o `calcula` del `PATH`; el timeout es de 30 segundos por respuesta.

Dos notas de ese acuerdo: el antecesor de calcula es la función `psicro` de los sceptre-simusol viejos (FUNCTIONs Fortran enlazadas al generado, con selectores numerados para elegir qué propiedad devolver); y `calcula --machine` no combina hoy con `--modelo`, así que `CALCULA('X.M.dsa', …)` desde el deck espera ese camino (o traducir el `.M.dsa` antes). El brazo de la fase 2 se probó solo (03/11/2026): un programa que llama `calcula_valor` con un sistema compilado, pide dos salidas con los mismos datos (un solo `EVAL`, por el caché) y una tercera con un dato cambiado (reevalúa).

Quedan del lado del protocolo, anotadas en `pasos/PENDIENTES`: `SEMILLAS` e `ITER` (arrancar el Newton de calcula desde la predicción del integrador, como hace siru) y las

sensibilidades por el protocolo (`SENS1`: que el módulo entregue también derivadas, hoy sólo una idea con nombre).

6 La corrida: dinámico, reentrante, sin topes

6.1. El circuito como `.so`

La fase 2 es un binario fijo, `bin/sceptreDU`, enlazado una vez con toda la biblioteca. El programa generado se compila con `-fPIC -shared` a `CASO.so` y `sceptreDU` lo carga con `dlopen`. No se enlaza un ejecutable por circuito.

El par se guarda junto al `.d` y se reutiliza: `sc1` deja al lado del deck `CASO.so`, `CASO.datasave`, `CASO.acsave` si hubo y `CASO.huella` (los md5 del `.d`, de las bibliotecas de modelos —la instalada y las del usuario—, de las fuentes de sceptreDU, de `sceptreM` y de la usrlib propia si se dio una). Si las huellas coinciden, `sc1` no corre la fase 1 ni gfortran: lo dice en el log y va directo a `sceptreDU`. Cambiar datos o usos no recompila; cambiar el modelo, sí (capítulo 3). `-sodir DIR` elige dónde queda el par y `-rehacer` lo ignora.

El `.so` sabe de dónde salió: lleva `SC1_ORIGEN` con las huellas, y `sceptreDU` rechaza con los códigos 83–85 lo que no coincide (cargar falló, le falta una entrada, o la huella no es la de este deck: rehacerlo).

6.2. Reentrancia e instancias

La fase 2 es reentrante: el estado de una corrida vive en una instancia, no en variables estáticas (ya no se compila con `-fno-automatic`; lo que una rutina recuerda lo declara un `SAVE` con lista, también en el generado). Varias instancias conviven en el mismo proceso, y sobre eso se apoyan el servidor y los modelos declarados (un `.so` por modelo cuando el deck declara `OBSERVABLES`, `VARIABLES` o `LINEALIZADO`).

6.3. El servidor y los lotes

Para correr muchos circuitos sin un proceso por cada uno, `sceptreDU --servidor` lee trabajos por la entrada estándar (`DIRECTORIO CASO.so [VAR=VALOR…]`, uno por renglón) y contesta `DIRECTORIO CODIGO`. Cada trabajo corre en un proceso hijo: un fallo contesta su código y el servidor sigue. Desde `sc1`, sin armar renglones: `sc1 -lote CASO1.d CASO2.d …` corre cada deck como una corrida común (la fase 1 de todos la hace un solo `sceptreM --lote`) y todas las fases 2 van por un solo servidor; al final escribe `CASO CODIGO` por deck y sale con el primer código distinto de cero.

6.4. El latido

Las dos fases llevan un latido: sus lazos principales cuentan que siguen vivos. Si la cuenta no se mueve en `SCEPTRE_SIN_AVANCE` segundos (300 por omisión; 0 lo apaga), la fase 2 *guarda lo hecho* –los puntos de salida que hubo, el `.dirac`, el listado– y sale con 118 diciendo dónde estaba, `TIME` y `FREQ`. En el transitorio corta también si `TIME` se queda quieto o pasa de largo `STOP TIME` sin terminar. La fase 1 tiene además un tope de tiempo total, `SCEPTRE_FASE1_TOPE` (1800 s), porque contar latidos no ve un lazo que late sin avanzar.

`sc1 -sinlatido` apaga los dos cortes, para la corrida que se sabe larga. El ejemplo que se cuelga a propósito es `corpus/cuelga.d`.

6.5. Memoria dinámica

Las tablas internas crecen con el problema: los topes fijos del SCEPTRE histórico desaparecen (quedan los pocos que son límites de formato, como los 999 de la formulación vectorial, con su código propio). La variable de prueba `SCEPTRE_COTAS_MINIMAS` arranca las tablas en 1 y las hace crecer todo el tiempo: el resultado tiene que ser idéntico, y así se ejercita el crecimiento (capítulo 10). La capacidad `TOTAL` del lenguaje está en el capítulo 4.

7 El álgebra: LAPACK, BLAS y UMFPACK

El álgebra de la fase 2 va por omisión por bibliotecas externas: LAPACK/BLAS para lo denso (inversión, sistemas lineales, autovalores, matriz por vector) y UMFPACK para el `SOLVE` disperso del método de Gear. Los métodos propios históricos quedan intactos detrás de una llave:

```
SCEPTRE_ALGEBRA=propia sc1 CASO.d   # los metodos propios, bit a bit
sc1 CASO.d                          # LAPACK/UMFPACK (el normal)
```

- La llave es *del proceso*: no entra en la huella del `.so` (el generado no cambia; cambian las adentros de sceptreDU).
- Los diagnósticos son *los mismos* por los dos caminos (códigos 111, 112, 116/117, 120/121 y sus mensajes); si UMFPACK falla en algo, `SOLVE` cae al camino propio con la misma entrada.
- Con LAPACK el resultado es *equivalente, no bit a bit*: el corpus compara con el umbral de cada caso. Con `propia`, bit a bit lo de siempre.
- La reproducibilidad pide un hilo de OpenBLAS: `sc1` exporta `OPENBLAS_NUM_THREADS=1` si no viene puesto.

7.1. La matriz en datos: `SCEPTRE_MATRIZ=numerica`

En los circuitos grandes la matriz de conductancias generada como texto Fortran crece sin remedio: la matriz simbólica densa es cuadrática en texto y un caso real dio un fuente generado de 3 GB que no se podía ni compilar. Con `SCEPTRE_MATRIZ=numerica` la fase 1 vuelca los términos ralos de la matriz a `matsave.dat` y el programa generado la arma numéricamente en la corrida: ese mismo caso baja a un generado de unos 3 MB y corre entero. Es una representación rala de los *términos*; el solver sigue siendo el denso (el álgebra rala de verdad es UMFPACK, arriba).

7.1.1. Por qué crecía, y qué se volcó

La fase 1 escribía cada entrada de las matrices del circuito como suma simbólica: $O(n)$ términos por entrada y $O(n^2)$ entradas, o sea un texto que crece como n^3. Pero todo lo que ese texto expande ya estaba en tablas de la fase 1, porque la matriz es

$$XM = D + B\,\mathrm{diag}(g)\,B^t$$

con B la matriz de incidencia empaquetada en bits. Así que la fase 1 vuelca, por matriz, un registro al `datasave`/`matsave` —dimensiones, banderas, la B rala y el índice en `valu()` de cada elemento con su marca de recíproco— y en el generado quedan `CALL SC2_ARMA_MATRIZ`, `SC2_ARMA_VECTOR` y `SC2_ECU` donde había $n^2/2$ asignaciones. Entró lo mismo para los vectores de `BVPSCN` (incluso con elementos variables: se repite el barrido sólo para registrar

dependencias y el buffer se tira), las diagonales y la recombinación de `SCNEQS`; `MRGEQS` no lo necesita, su texto ya era $O(n)$. **No queda nada que caiga al texto por n^2 o n^3**: el texto de siempre vale para lo que es $O(n)$.

Lo medido en el caso grande (`resescalera1200`, 1200 resistores en serie y 1200 a tierra): el generado pasó de ∼3,2 GB a 37 930 renglones (∼3 MB), la fase 1 de 4,4 minutos a 3,2 s, gfortran de no poder compilar (el OOM mataba a `f951` con 6,7 GB) a 0,74 GB de RSS, y el deck **corre entero**: código 0, 1:51 de pared. Dos lecciones que conviene no volver a aprender: la escalera no entra por `RESEQS` sino por `RESINI` (la inicialización del transitorio arma `XMRG` una vez si las R son constantes), y el `datasave` son tarjetas de 72 columnas útiles —con `(8i10)` el octavo entero de cada registro se recortaba—. Y el defecto que pescó el corpus: el índice en `valu` *no* se calcula, porque `valu` se liga por posición de una tabla que después se reordena; quedó una foto de la tabla al ligar y la resolución por nombre contra esa foto.

7.2. Lo medido, y dónde paga cada cosa

Los envoltorios conservan la interfaz y el papel de cada rutina —`INVERS` (`dgetrf`+`dgetri`/`dgetrs`, con el mismo criterio de pivote $< \epsilon$ para el 111), `LINSOL`, `MXMULT` (`dgemm` en los cuatro casos densos; los diagonales quedan propios), `LSNEW` (`dgeev` en lugar de la cadena Hessenberg propia), `INVRT` (`zgetrf`+`zgetri`, la inversión compleja del AC) y el `SOLVE` disperso (UMFPACK)—, así que el mismo `.so` sirve con y sin llave. El matvec del generado es el caso distinto: `DOGEN` emite *siempre* `CALL SC2_MATVEC` y el envoltorio decide en corrida (sin llave calca el lazo del texto, bit a bit).

- **Donde paga mucho**: una inversión densa grande —banco suelto $n = 1199$: 23,8 s el Gauss-Jordan propio contra 0,24 s `dgetrf`+`dgetri`, cien veces— y los AC grandes (`paralelo500acd`: 4,1 s a 1,7 s de CPU de simulación).
- **Donde paga poco**: el corpus chico (18 generados llaman `INVERS` con n entre 1 y 9: el atajo 1×1 propio y el overhead de LAPACK empatan), y `resescalera1200`, cuya `XMRGI` es diagonal: su álgebra es sólo el matvec, donde `dgemv` ahorra unos 8 s de 121.
- **La factorización persistente de UMFPACK** (la numérica de la última factorización guardada como estado del circuito) es la que más rinde en los Gear ralos: `c11` 6,18 → 1,99 s, `dispgrande` 2,92 → 1,03 s, `escalera2000` 0,69 → 0,16 s.

La certificación fue el corpus en las *cuatro* combinaciones (álgebra propia o externa × matriz texto o numérica): 105 patrones idénticos en las cuatro —bit a bit con `propia`, y con el externo los cuatro casos que se mueven caen dentro de su umbral medido—. Por eso el juez no es el bit a bit entre álgebras: LAPACK suma en otro orden. Un detalle anotado: con `dgeev`, `c08` pierde el renglón del aviso de `CHECKS`, que sólo sale si el error de autovectores pasa de 10^{-5} —los de `dgeev` son más precisos—; el aviso fue a stderr y su patrón se regeneró. Y el *frontal del AC* que el inventario pedía desacoplar no existe como solver emitido: lo que la fase 1 emite son las fuentes y los productos (`KKCALC`, `CDCALC`, `AGDIM`); el solver AC es la cadena de la fase 2, que ya va entera por LAPACK.

8 Las llaves del proceso

El comportamiento se gobierna con variables de entorno `SCEPTRE_*` y opciones de `sc1`, sin recompilar. Son llaves *del proceso*: no entran en la huella del `.so`. Las leen las dos fases, así que valen con cualquier lanzador.

`SCEPTRE_LOG` el archivo del listado único (en append; sin ella, `sceptre.lis`).

`SCEPTRE_PARAR` `1` junta todos los errores del deck (omisión); `2` para en el primero.

`SCEPTRE_VERBOSE` `1` sólo errores y el renglón de salida; `2` el listado completo (omisión); `3` con detalle interno.

`SCEPTRE_SIN_AVANCE` segundos sin avance antes de que el latido corte con 118 (300; `0` lo apaga).

`SCEPTRE_FASE1_TOPE` tope total de la fase 1, en segundos (1800; `0` lo apaga). `sc1 -sinlatido` apaga los dos.

`SCEPTRE_ALGEBRA` `propia` para los métodos numéricos históricos (capítulo 7).

`SCEPTRE_MATRIZ` `numerica` para armar la matriz en la corrida (capítulo 7).

`SCEPTRE_TABLAS` / `--tablas` `tabular2` (omisión) o `sceptre`, el interpolador histórico.

`SCEPTRE_DATOS` nombres y valores que pisan a los del deck en la fase 2, sin regenerar el circuito (códigos 88/89 si algo no existe o no se pudo leer).

`SCEPTRE_CALCULA` el ejecutable de calcula para la función `CALCULA` (omisión: el del `PATH`).

`SCEPTRE_NO_CONF` no leer la configuración del usuario (`~/.config/sceptre/sceptre.conf`); la usa el corpus para que una configuración no cambie los patrones.

`SCEPTRE_COTAS_MINIMAS` sólo para probar: las tablas arrancan en 1 y crecen todo el tiempo; el resultado debe ser idéntico.

`SCEPTRE_HOME` dónde está el árbol, para los guiones; sin ella, lo deducen de su propia ubicación.

`TMPDIR` / `GFORTRAN_TMPDIR` dónde van los temporales de la fase 1 (capítulos 2 y 9).

`OPENBLAS_NUM_THREADS` `sc1` lo pone en 1 si no viene, por la reproducibilidad (capítulo 7).

Hay además variables internas de la cadena (`SCEPTRE_SO`, las huellas `SCEPTRE_HUELLA_*`) que `sc1` administra solo; un lanzador propio las encuentra en `INVOCACION-SCEPTRE.md`.

9 Diagnósticos

9.1. El renglón de salida y los códigos

Antes de cada salida distinta de 0, las dos fases escriben –en el listado y por la salida de error– un renglón de seis campos fijos separados por `|` :

```
SCEPTRE: SALIDA | 124 | 21:30:44 | trprk.f90:116 | TRPRK | las derivadas piden un paso
    menor que el minimo
```

el literal, el código con que va a salir el proceso, la hora, el archivo fuente con su renglón, la rutina y el mensaje en una línea. **Para analizar una corrida alcanzan dos cosas**: el código que devolvió el proceso y el último renglón `SCEPTRE: SALIDA` del log. Cuando un fallo anota y sigue, puede haber varios renglones; el proceso sale con el código *más alto* anotado.

Los códigos son uno por punto de salida y caen en el rango de su familia: archivos y base de resultados (5–9), tablas desbordadas (10–29, casi vacío con la memoria dinámica), área de trabajo (30–39), no se encontró algo del deck (40–69), estados que no se saben manejar e internos (70–89), lo no implementado y la memoria (90–99), el deck y el guion (101–109), los fallos de la simulación (111–128). Ninguno pasa de 125, porque de 126 para arriba el shell los confunde con los suyos. La tabla número por número está en el apéndice A, con las dos reglas que la gobiernan. Cuando esto se construyó (14–21/10/2026) *todo salía con 0* —un deck rechazado y una corrida buena eran indistinguibles para el shell— y los primeros códigos destaparon dos fallas de simulación silenciosas que el corpus venía aceptando (d04 agota pasadas en las condiciones iniciales; dpadd pide un paso menor): conviene saberlo al leer sus listados.

9.2. El origen de cada diagnóstico

`sc1` guarda el nombre del archivo original en `input.dat.origen` junto a cada entrada preparada, y las fases añaden a stderr `SCEPTRE: ORIGEN | archivo | mensaje`: todo diagnóstico puede nombrar al verdadero culpable aunque la entrada haya sido traducida o expandida.

9.3. El mapa del programa generado

`sc1` deja `CASO.origen.json`: relaciona las asignaciones del Fortran generado con las ramas, parámetros y funciones del fuente. Si la fase 2 falla, stderr añade `SCEPTRE: MAPA DE ECUACIONES | ruta`. Cuando `LINSOL` o `SOLVE` detectan una matriz singular guardan la matriz, y el análisis amplía el mapa con el rango estimado, las combinaciones dependientes –con sus filas y líneas del fuente–, la dirección indeterminada y, para la matriz de integración, qué estado es cada coordenada. Es una relación conservadora de dependencias, no una atribución de culpa a una ecuación única.

9.4. Los avisos de recursos

La fase 1 avisa en el listado –una vez, sin cortar– cuando a los temporales les queda poco lugar (`CUOTA`): en las máquinas nuevas `/tmp` vive en RAM y con cuota de usuario, y un circuito grande la agota con síntomas engañosos (parece un cuelgue y es el runtime reintentando una escritura que no va a entrar). La salida es correr con `TMPDIR` en disco (capítulo 2). El código 96 es «no queda lugar donde la corrida escribe».

10 Verificación: el corpus

El árbol se verifica contra un corpus de más de cien casos con patrón, todos en el formato `.dsa`: el par `CASO.M.dsa` + `CASO.DU.dsa`, que la fase 1 lee directo. (Los mismos circuitos como `.d` se conservan en `viejos/corpus`, y los decks que prueban la forma original de SCEPTRE en `lanzador/corpus/originales`; ninguna prueba del árbol los usa.) Los `.dsa` salieron de esos `.d` con `tools/pasar-a-dsa`, que convierte un árbol entero comprobando la vuelta y sin pisar nada escrito a mano. Cada caso tiene `CASO.caso0.patron` (la salida filtrada de una corrida de referencia: el `.ls2` más el `.sim`, o el `.ls1` en los casos de error a propósito) y `CASO.caso0.codigo` (`fase1=N fase2=M`, los códigos esperados). Nada se da por bueno si los patrones no salen idénticos –o dentro del umbral propio del caso, donde el álgebra externa da equivalente y no bit a bit.

```
CORPUS=... ARBOL=$N SRCARBOL=$N FC=gfortran MODO=estatico  sh $N/tests/check-corpus.sh
CORPUS=... ARBOL=$N SRCARBOL=$N FC=gfortran MODO=dinamico sh $N/tests/check-corpus.sh
```

Los dos modos corren el corpus entero; el dinámico ejercita además la reutilización del `.so`. Un patrón se regenera con `corpus/hacer-patron ARBOL CASO X`, que corre el caso desde cero (borra el log en append: un patrón sale de *una* corrida) y filtra lo que cambia siempre (horas, números de línea, tiempos de CPU).

Piezas del corpus que conviene conocer: `CASO.base` (el caso corre sobre los restos de otro: `CONTINUE`/`RE-OUTPUT`), `CASO.entorno` (variables sólo para ese caso; lo usa `cuelga`, el que se cuelga a propósito para probar el latido), los casos de error a propósito (`ciilegal`, `lazofuen`, `sinelementos`, …, que ejercitan los códigos) y `SCEPTRE_NO_CONF`, para que la configuración del usuario no cambie los patrones. El corpus es el método desde el principio y es lo que cubre también los pasos viejos: de los casi 300 `pasos/*.diff` hay 48 —los primeros, de cuando la forma de anotar era otra, más las tandas de los COMMON— que no traen su renglón de medición; lo que les falta es la *anotación*, no la verificación: ninguno entró sin que el corpus diera idéntico, y el árbol de hoy, con los 48 adentro, da sus patrones en los dos modos. La variable `SCEPTRE_COTAS_MINIMAS` (capítulo 6) repite el corpus ejercitando el crecimiento de la memoria dinámica.

11 Construcción e instalación

El árbol se construye **con autotools, y nada más**. Los `Makefile` planos de cada directorio —los del trabajo diario hasta octubre de 2026— se fueron cuando automake pasó a compilar de verdad: lo que queda de ellos es la costumbre de `make -C` sobre el directorio de compilación. Los guiones (`sc1`, `sd2`, `exa.sh`) son bash directo y calculan solos dónde está el árbol: `SCEPTRE_HOME` o el padre del directorio del guion; y `sc1` busca en `bin/` y `lib/` *antes* que en el árbol, así que una copia desprolija deja midiendo con el ejecutable anterior (`hacer-patron` del corpus lo avisa si los ve desparejos).

11.1. Autotools: configure, automake y libtool

Para distribuir e instalar como paquete, el árbol también se construye con autotools: `autogen.sh` genera el `configure` (`configure.ac`), automake arma los `Makefile` desde los `Makefile.am`, y libtool construye `libsceptreDU.la`. El `configure` busca las bibliotecas (tabular2, LAPACK/BLAS, UMFPACK) y `make dist` empaqueta las fuentes (los Makefiles artesanales viajan en `EXTRA_DIST`: son los del trabajo diario). La huella de sceptreDU (`sceptreDU.huella`, el md5 de sus fuentes) se escribe al construir y es la que protege la reutilización del `.so` (capítulo 6). El detalle del camino automake está en el agregado 13.11 y el empaquetado con summapack en el 13.12; la instalación normal de un proyecto Simusol va por `sp`.

11.2. La instalación a través del tiempo: el caso de Dia

La instalación de Simusol (y de SCEPTRE con él) pasó por tres épocas, y la diferencia explica una decisión de diseño:

1. **Al principio**: se bajaba `simusol-last.tgz` y se corría `sudo make install`, que copiaba a mano formas, hojas y guiones.

2. **Después hubo que recompilar Dia**: el menú Simusol adentro de Dia va por PyDia, el plugin Python de Dia, y los Dia de las distribuciones venían sin ese plugin compilado (y, cuando el menú pasó a Python 3, con el plugin de Python 2). Para tener el menú había que reconstruir Dia habilitándolo.

3. **Hoy ya no hace falta**, por dos razones. La primera: los Dia actuales de las distribuciones (0.98 en adelante) ya traen el plugin Python 3 habilitado, así que el menú se instala copiando `simusol.py` y nada más. La segunda, la de fondo: la integración con Dia quedó *débil a propósito* –las formas y las hojas son datos XML que sólo se copian, el lanzador `dia-simusol` arranca el Dia común de la distribución, y todo el procesamiento del `.dia` ocurre *afuera* de Dia, sobre el archivo guardado–. No hay código nuestro compilado adentro de Dia, así que no hay nada que recompilar cuando Dia cambia. La instalación va por paquetes (`sp`, summapack, agregado 13.12); y un usuario sin root instala desde fuentes en su home, con `sp --para fuentes --prefijo ~/.local install PROYECTO`: `make install` con ese prefijo, sin sudo. Los tres prefijos –`/usr` con paquete, `/usr/local` desde fuentes, `~/.local` sin root–

son los que el resto del proyecto (por ejemplo, el menú Documentos de Dia) sabe mirar.

11.3. El MANIFIESTO

`nuevo/MANIFIESTO` lista archivo por archivo qué es fuente del árbol: es la lista de embarque de `make dist` y la referencia de qué se versiona. Un fuente nuevo se agrega ahí.

12 Cómo pedirle los comportamientos de antes

Todo el libro describe el comportamiento normal: dinámico, reentrante, álgebra por bibliotecas, tablas con tabular2. Cada pieza conserva su camino histórico detrás de una llave:

Modo estático `sc1 -estatico CASO.d`: el programa generado se enlaza en un `sceptreDU` propio del caso, como antes de los `.so`.

Álgebra propia `SCEPTRE_ALGEBRA=propia`: los métodos numéricos históricos, bit a bit con el SCEPTRE de referencia.

Interpolador histórico `--tablas sceptre` (o `SCEPTRE_TABLAS=sceptre`): las `TABLE` como las interpolaba SCEPTRE.

Matriz generada como texto sin `SCEPTRE_MATRIZ`: la matriz va escrita en el Fortran generado, como siempre.

Sin latido `-sinlatido` (o `SCEPTRE_SIN_AVANCE=0`): ninguna vigilancia de avance.

El árbol `cuasioriginal` –el SCEPTRE de referencia con el que se generaron los patrones históricos– se conserva en el repositorio y sigue corriendo: es el testigo de compatibilidad.

Y para comparar contra el linaje: el lanzador de `lanzador` sabe lanzar los motores `sceptre-dinamico`, `sceptre-estatico` y `sceptre-viejo` (el SCEPTRE de `sceptresimusol`, que sigue estático), y los códigos de salida son los mismos en los dos árboles por decisión del autor: quien los llama no necesita saber con cuál habla.

13 Agregados documentados durante la traducción

Estas secciones se escribieron mientras se traduía el manual histórico (`docs/manual-es/agregados`): documentan cosas nuevas del proyecto que no correspondían a aquel libro. Se incorporan acá tal cual; a medida que los capítulos de este manual las absorban, se irán fundiendo.

13.1. Capacidad total y capacidad diferencial

Origen: cambio del proyecto. El modificador `TOTAL` se incorporó a esta implementación el 20 de octubre de 2026. No forma parte de la sintaxis del árbol `cuasioriginal`. Los archivos del corpus citados aquí son pruebas nuevas de esa ampliación.

Sin modificador, el valor de un capacitor es la capacidad diferencial:

$$i = C_{\mathrm{d}}(v)\,\frac{\mathrm{d}v}{\mathrm{d}t}.$$

Con `TOTAL`, el usuario proporciona la capacidad total o secante, definida por $q(v) = C_{\mathrm{t}}(v)v$. El programa obtiene la capacidad que necesita el integrador:

$$C_{\mathrm{d}}(v) = \frac{\mathrm{d}q}{\mathrm{d}v} = C_{\mathrm{t}}(v) + v\frac{\mathrm{d}C_{\mathrm{t}}}{\mathrm{d}v}.$$

No son dos notaciones intercambiables para una misma función numérica.

Ejemplo de la ampliación, tomado de `totalvar.d`:

```
ELEMENTS
  C1, 2-0 = TOTAL Q1(VC1)
FUNCTIONS
  Q1(X) = (1.+X*X)
```

Aquí $q = v + v^3$, por lo que el integrador utiliza $1 + 3v^2$. La comparación con sintaxis heredada está en `totalequiv.d`:

```
ELEMENTS
  C1, 2-0 = Q1(VC1)
FUNCTIONS
  Q1(X) = (1.+3.*X*X)
```

Al pedir `C1` en `OUTPUTS` se obtiene la capacidad diferencial, no la función total introducida. Para una constante ambas capacidades coinciden: `totalcte.d` y `difcte.d` comparan `TOTAL 2` con `2`.

La conversión actual utiliza una diferencia centrada de q, con desplazamiento

$$h = 10^{-6}\,\mathrm{máx}(1, |v|).$$

La diferenciación numérica no elimina la necesidad de que la función sea adecuada para la simulación. Las variantes `totalvarimp.d` y `totalvarvec.d` prueban, respectivamente, integración implícita y formulación vectorial.

Fuentes: `sceptre2novender/pasos/capacidad-total.diff`; rutinas `ELMPRS` y `EQFORM` del árbol `nuevo`.

13.2. Restricciones y reconocimiento del modificador TOTAL

Origen: cambio del proyecto. Estas reglas pertenecen a la ampliación `TOTAL`; no son restricciones nuevas para los elementos escritos con la sintaxis heredada.

- El modificador sólo se admite en un capacitor `C...`.
- La ley debe depender únicamente de la tensión del propio capacitor, además de constantes y las funciones o tablas admitidas por la comprobación del programa.
- No se admite una dependencia de `TIME` ni de otras tensiones de la red para esta conversión.
- `TOTAL` debe reconocerse como palabra separada al comienzo del valor. No debe confundirse con un nombre de tabla o función.

Prueba negativa nueva, `totalmal.d`:

```
R1, 1-2 = TOTAL 1
C1, 3-0 = TOTAL Q1(VC1,VR2)
C2, 2-0 = TOTAL Q2(TIME)
```

Los tres renglones son intencionalmente inválidos: el primero aplica el modificador a una resistencia; los otros introducen dependencias no admitidas. El diagnóstico específico es el mensaje 109, `TOTAL ONLY FOR A CAPACITOR DEPENDING ON ITS OWN VOLTAGE`. El número del mensaje del listado no debe confundirse con el código de terminación del proceso: la fase 1 acumula estos errores y rechaza la simulación.

Prueba nueva de compatibilidad con una sintaxis heredada, `totalnombre.d`:

```
ELEMENTS
  C1, 2-0 = TOTAL1(VC1)
FUNCTIONS
  TABLE OTAL1
    -2,13,-1,4,0,1,1,4,2,13
```

`TOTAL1` se interpreta como `T` más el nombre de tabla `OTAL1`. No activa el modificador. Este caso conserva el significado que tenía la notación abreviada de tablas antes de añadir `TOTAL`.
Fuentes: `sceptre2novender/pasos/capacidad-total.diff`, `sceptre2novender/nuevo/sceptreM/elmprs.f90` y `sceptre2novender/nuevo/sceptreM/error.f90`.

13.3. Reparaciones de la convolución

El proyecto recuperó caminos de ejecución deshabilitados y corrigió defectos de preparación y cálculo de los elementos de convolución. El ejemplo heredado `a06b.d` fue reparado y ajustado. Estas intervenciones no crean las instrucciones `FCONVE` y `FCONVJ`; su explicación heredada está en `masexplicaciones/03-elementos-convolucion.tex`. Las reparaciones no garantizan estabilidad para cualquier paso de integración.
Fuentes: este agregado; `STCONV`, `CVPREP`, `CNVCOM`, `RES` y `CVYCLC`.

13.3.1. Cómo se revivió, y por qué pide paso fijo

El modo convolución estaba **muerto** en este árbol y defectuoso en todos los anteriores. Se arreglaron seis cosas, en el orden en que aparecieron: un *segfault* de la fase 1 (tres rutinas de convolución llamaban a `WTSPTP` con la interfaz vieja de cuatro argumentos, marcadas «no revision»); el armado, que dejaba la `FUNCTION FCONVY` dentro del `PROGRAM` (`XF90` clasificaba su cabecera como declaración); cuatro *stubs* en la fase 2 que abortaban con «convolution mode not supported» *tapando* el código real, entero, que estaba debajo; y la **cinta de núcleos**, que no existe en ningún árbol porque el SCEPTRE original la recibía de afuera (JCL). Su formato se confirmó contra el apéndice F del manual y contra `RDCONV`: por núcleo, el identificador, $h(0^+)$, el valor en *frecuencia cero* $H(\omega{=}0)$ —no una «resistencia de entrada»: para un RLC paralelo vale 0, porque la bobina es corto en continua—, el largo, y la tabla con *tiempo negativo*. El núcleo de las cajas de `a06a` es analítico,

$$h(t) = \tfrac{1}{C}\, e^{-at}\left[\cos(\omega_d t) - \tfrac{a}{\omega_d}\sin(\omega_d t)\right], \quad a = \tfrac{1}{2RC},\ \omega_d = \sqrt{\tfrac{1}{LC} - a^2}$$

y quedó generado en `corpus/fort.12` (tres núcleos, 201 puntos).

Los otros dos eran de *estabilidad*, y son la lección que conviene conservar. El elemento de convolución se resuelve con un companion cuya resistencia equivalente es **proporcional al paso** ($R_{eq} = (2h(0) + h(\Delta t))\Delta t/6$, la fórmula F.21 del manual, verificada derivándola): cuando el control de error reduce el paso, R_{eq} se achica, las corrientes crecen, el error crece, y la espiral termina en un corto y el aborto. Con el control de error anulado y paso fijo grande pasa lo contrario: `a06b` tiene un estado rápido ($L_1 = 25$ H con ∼6,2 kΩ: $\tau \approx 4$ ms) y un paso de 15 ms deja $h\lambda = -3{,}7$ *fuera* de la región de estabilidad del RK explícito, cuyo factor de crecimiento en $-3{,}7$ es justamente el 3,6 por paso que se medía. Las dos divergencias son los dos lados del mismo límite. La receta correcta —la misma que advierte siru: «el paso adaptativo no ve el error de la convolución»— es paso fijo *por debajo* del límite del estado rápido: 0,005 en `a06b` y `cmpac`, con el `INPUT FUNCTION BUFFER` en 1200 (a ese paso la corrida pide 941 puntos de historia).

La validación, punto a punto: con eso `a06b` corre entero y empalma con siru en la misma grilla —941 puntos, ocho señales, peor diferencia relativa $6{,}2 \cdot 10^{-4}$—, y siru estaba validado contra `a06a`, el mismo circuito con las cajas RLC explícitas: la cadena `a06a` = siru = sceptre cierra. *La convolución del SCEPTRE quedó validada de punta a punta en PC por primera vez.* Lo que falta está en `pasos/PENDIENTES`: avisar o forzar paso fijo cuando hay elementos de convolución (hoy quien no sepa la receta cae en la espiral sin ninguna pista), la forma recursiva para núcleos que son suma de exponenciales (siru la usa y evita el $O(n^2)$), y ejercitar la compresión de la tabla de historia.

13.4. Corrección de lectura de los núcleos de convolución

Corrección del proyecto: se usa `G20.0` para leer los valores. Con el antiguo `G20.8`, un campo sin punto decimal podía interpretarse con un factor 10^{-8}. Los campos siguen teniendo ancho fijo; no basta con escribir números separados por espacios sin respetar los anchos.

El formato y la interpretación física heredados se explican en `masexplicaciones/04-nucleos-convolucion.tex`.

Fuentes: `sceptre2novender/nuevo/sceptreDU/rdconv.f90`, este agregado y `sceptre2novender/corpus/fort.12`.

13.5. Dimensionamiento del ejemplo de convolución reparado

Ejemplo heredado ajustado por el proyecto: `a06b.d` usa un paso próximo a 0,005 hasta $t = 4{,}7$. Sin compresión, el orden de magnitud del historial requerido es

$$N \simeq 1 + \frac{T}{\Delta t},$$

pero deben dejarse posiciones adicionales para la inicialización y las evaluaciones del algoritmo. El ejemplo actual usa 1200 posiciones; la documentación de la reparación registra que 900 no alcanzaban para esa corrida.

El significado de ambos controles heredados está en `masexplicaciones/05-buferes-convolucion.tex`.
Fuentes: `CVPREP`, `CVYCLC` y este agregado.

13.6. Corrección de la desactivación de la compresión

Corrección del proyecto: en la implementación actual, `COMPRESSION COUNT = 0` desactiva la compresión. Se corrigió un camino heredado que, con cero, podía invocar la rutina con una ventana vacía y añadir puntos espurios.

La prueba `cmpac.d` deriva del ejemplo heredado. El algoritmo y sus controles están explicados en `masexplicaciones/06-compresion-convolucion.tex`.
Fuentes: `sceptre2novender/nuevo/sceptreDU/cmpac.f90` y `sceptre2novender/nuevo/sceptreDU/cvyclc.f90`.

13.7. Separación interna de la selección vectorial

Cambio interno del proyecto: la bandera `lvecfrm` separa conceptualmente la formulación del umbral, aunque su selección inicial sigue comparando el tamaño de la red. Otras decisiones de matrices y cálculo también usan ese umbral. No es una garantía de mayor velocidad ni de convergencia; cero tampoco debe presentarse como un sustituto seguro de uno.

`funcxvec.d` comprueba además el uso de `X` como parámetro formal tras una reparación del proyecto.

El control heredado se explica en `masexplicaciones/08-ecuaciones-vectoriales.tex`.
Fuentes: `sceptre2novender/nuevo/sceptreM/setup.f90`, `sceptre2novender/nuevo/sceptreM/common_tables.f90`.

13.8. Fin de archivo en la lectura del monitor de terminal

Reparación del proyecto, 22 de octubre de 2026: al terminar, el monitor intentaba leer un retorno de carro. Si el lanzador había cerrado la entrada, el fin de archivo causaba un error de ejecución. Ahora esa lectura usa `IOSTAT` y tolera el fin de archivo. El ejemplo es una prueba del arreglo, no una nueva instrucción del deck.

El funcionamiento heredado del monitor se explica en `masexplicaciones/09-salida-terminal.tex`.
Fuentes: `sceptre2novender/nuevo/sceptreDU/intplt.f90`; defecto 56 de `sceptre2novender/pasos/DEFECTOS`.

13.9. Capacidad histórica y ampliaciones de la implementación actual

Origen: cambios del proyecto en almacenamiento y generación de código. La tabla de capacidad del manual describe la versión histórica que documenta. No es una prueba de que esos mismos máximos sigan vigentes en esta implementación ni de que toda cota haya desaparecido.

Recurso	Referencia histórica	Ejemplos nuevos del corpus
Elementos	300	`paralelo1200.d`: 2401; `dispgrande.d`: 14001.
Parámetros definidos	200 en la tabla del manual	`dpgrande.d`: 250; `dpmil.d`: 1200.
Salidas	100	`outgrande.d`: 120; `salidas720.d`: 720.
Índices generados	Campos internos de pocas cifras	`escalera2000.d` y variantes ejercitan números superiores a 999.
Matrices	Estructuras y direcciones antiguas	`dispgrande.d` y `dispdensa.d` prueban tamaños de almacenamiento mayores que 32767 posiciones.

Los ejemplos grandes son pruebas de regresión de las ampliaciones, no nuevas formas de escribir resistencias o capacitores. Los nombres y los elementos siguen las reglas del lenguaje; cambian las estructuras que los almacenan y las anchuras con que se generan índices y números en el Fortran.

Los tamaños anteriores describen la intención y el contenido de los archivos; no constituyen una certificación nueva de que todos hayan terminado correctamente con cualquier compilación. Los resultados dependen de la memoria disponible, la topología, el método de cálculo y las restricciones todavía presentes en cada camino.

El directorio Dirac ampliado permite registrar más entradas, pero no elimina por sí solo todos los límites del lector de resultados, de los nombres o de las corridas. Tampoco un número elevado de salidas mejora la exactitud del integrador.

Fuentes: el agregado 13.31, `sceptre2novender/pasos/tope-999-pasar.diff`, módulos de tablas del árbol `nuevo` y archivos citados del corpus.

13.10. Validación de TIME, FREQ y parámetros complejos

Origen: restricciones del lenguaje y diagnósticos añadidos o adaptados por el proyecto. Una entrada nueva del corpus puede comprobar una restricción antigua. Los mensajes 96–108 del árbol actual se incorporaron durante la modernización; no todos representan prestaciones nuevas.

Caso de prueba nuevo	Situación intencional	Mensaje de la fase 1
`timeargac.d`	`TIME` como argumento en AC	102: uso de tiempo en un argumento de AC.
`timeenac.d`	`TIME` pedido como salida AC	103: tiempo solicitado en AC.
`freqargtiempo.d`	`FREQ` como argumento fuera de AC	104: frecuencia en un argumento fuera de AC.
`freqcorto.d`, `freqentiempo.d`	Frecuencia pedida en un transitorio	105: frecuencia solicitada fuera de AC.
`wentiempo.d`, `wfuera.d`	Parámetros `W` en un análisis no AC	100 y/o 101, según la declaración y la salida solicitada.
`acsinelementos.d`	AC sin red declarada	108: no puede realizarse el análisis AC.

Los casos anteriores no deben presentarse como recetas válidas. Por contraste, `wenac.d` comprueba una declaración compleja en su contexto AC:

```
DEFINED PARAMETERS
  W1 = (1.,2.)
RUN CONTROLS
  RUN AC
```

El resto del archivo proporciona la red, las salidas y el barrido de frecuencias.

El mensaje del listado y su severidad describen cómo se trata cada situación. Un aviso puede permitir que el procesamiento continúe; un error de entrada puede acumularse con otros antes de rechazar la simulación. El número del mensaje no es el código de salida del proceso: por ejemplo, el mensaje 101 y una terminación de la fase 1 con código 101 pertenecen a numeraciones distintas.

Fuentes: `ERROR`, `ARGCKR` y comprobaciones de salidas de la fase 1; el apéndice A.

13.11. Construcción con Autoconf, Automake y Libtool

Origen: infraestructura incorporada por el proyecto. Los Makefile tradicionales se conservan para el trabajo directo. El sistema GNU añadido compila los fuentes: no es un envoltorio que delega la compilación en aquellos Makefile.

Autoconf: obtiene `configure` de `configure.ac` y comprueba compiladores y opciones.

Automake: obtiene las reglas de construcción desde los `Makefile.am` de la raíz, `sceptreM`, `sceptreDU`, `sd2` y `tests`.

Libtool: construye las bibliotecas de fase 2 compartida y estática mediante `libsceptreDU.la`.

Ejemplo de construcción del árbol actual, desde una copia limpia de `nuevo`:

```
autoreconf -fi
mkdir build-manual
cd build-manual
../configure --with-corpus=/ruta/absoluta/al/corpus
make
make check
make DESTDIR=/tmp/sceptre-instalacion install
```

La última orden prepara una instalación bajo un destino de prueba; no instala en el sistema activo. El usuario debe elegir un destino vacío propio. `--prefix` permite cambiar las rutas finales previstas.

Conviene compilar fuera del árbol fuente para conservar sus Makefile tradicionales. Si quedan objetos de una compilación artesanal, VPATH puede encontrarlos y tratarlos como productos disponibles; el diagnóstico de `configure` advierte esta situación. Una distribución de fuentes limpia evita mezclar ambos juegos.

`make check` usa el corpus indicado; no lo incorpora por arte de magia al paquete fuente. `make dist` prepara la distribución y `make distcheck` comprueba construcción, pruebas e instalación de esa distribución bajo sus condiciones de ejecución.

Los modos de enlace requieren artefactos diferentes: `sceptreDU`, `libsceptreDU.so`, `sceptreDU.o`, `libsceptreDU.a`, los módulos Fortran y la biblioteca de usuario. No deben confundirse las bibliotecas con los circuitos compilados por el lanzador. Las opciones BLAS/-LAPACK declaradas en `configure.ac` no acreditan por sí solas un backend implementado. **Fuentes:** los `configure.ac` y `Makefile.am` actuales de `nuevo`; el capítulo de construcción como antecedente, con estados históricos que pueden haber cambiado.

13.12. Empaquetado de SCEPTRE mediante SummaPack

Origen: integración del proyecto. SummaPack organiza la construcción y distribución del paquete; no modifica la sintaxis del circuito por el hecho de empaquetarlo.

`sceptre2novender.summa`**:** descriptor con versión, descripción, dependencias, licencia y reglas de empaquetado. La identificación de base es `sceptre2novender`; el nombre de distribución usado por Autoconf puede ser diferente.

Manifiesto: determina qué fuentes se exportan a la copia limpia usada para construir. Evita tomar todos los productos y temporales del directorio de trabajo como si fueran fuentes.

`hacer-pkg`**:** ejecuta la cadena de exportación, preparación y empaquetado apoyándose en las funciones comunes de SummaPack.

Ejemplo de construcción del paquete actual, para mantenedores:

```
cd /ruta/al/proyecto/sceptre2novender/nuevo
./hacer-pkg
```

El guion verifica herramientas, exporta una copia limpia en un directorio temporal, ejecuta `autoreconf`, construye el paquete Debian e intenta obtener los otros formatos previstos por las funciones compartidas. Los artefactos efectivamente producidos se copian al directorio de origen. Que un formato figure entre los objetivos no significa que se haya construido correctamente en una ejecución concreta.

Las dependencias de construcción y ejecución son diferentes. `gfortran` también es necesario al ejecutar simulaciones porque el lanzador compila el Fortran del circuito. La preparación de los metadatos de paquete no reemplaza las pruebas del corpus ni demuestra que se hayan distribuido las bibliotecas y módulos requeridos por ambos modos de enlace.

Para instalar paquetes publicados, la vía del proyecto es `instalarsumm.ah`. No debe presentarse un comando interno de descarga como interfaz de instalación para el usuario. La disponibilidad de un paquete concreto depende del repositorio configurado y no se deduce de la mera existencia del descriptor local.

El descriptor y el guion contienen documentación histórica de pasos de modernización. Al documentar qué se instala realmente deben cotejarse con los `Makefile.am` vigentes, especialmente después del cambio a Libtool y a carga dinámica.

Fuentes: `sceptre2novender/nuevo/sceptre2novender.summa`, `sceptre2novender/nuevo/hacer-pkg` y los auxiliares de SummaPack que éste carga.

13.13. Tabular: lectura de resultados y conversión de tablas

Origen: integración del proyecto y ampliación del lector de Tabular. SCEPTRE produce sus resultados; Tabular puede leerlos y convertirlos sin volver a resolver el circuito. Este papel debe distinguirse de la interpolación de tablas usadas como datos de entrada.

Ejemplo de conversión de un resultado existente:

```
sceptre-tabular resultado.dirac --salida resultado.simul
```

`sceptre-tabular` también puede lanzar una instalación compilada de SCEPTRE y exportar sus resultados. Para convertir un binario ya existente no hace falta recompilar ni ejecutar la simulación. El programa usa `interpola --exportar-simul`; los detalles y opciones pertenecen al conversor, no al lenguaje `.d`.

El lector de `tabular2/src/pos4.c` admite el directorio antiguo y el formato POS4 con uno o varios registros de directorio. Detecta el orden de bytes y convierte los vectores `REAL*4` a valores de trabajo. Comprueba longitudes, regiones, nombres, corridas y el eje `TIME`/`FREQ`, en lugar de interpretar cualquier secuencia de bytes como una tabla.

Cambio del proyecto: el lector se amplió para aceptar el directorio extensible que escribe el SCEPTRE actual. Un lector antiguo puede aceptar archivos pequeños y fallar con uno que tenga más entradas de directorio; ese éxito parcial no prueba compatibilidad con el tercer formato.

Existe además `sceptre2simul` en el árbol de SCEPTRE. Su lector Python inspeccionado exige un directorio POS4 de un registro; no debe atribuirse automáticamente a ese programa toda la compatibilidad de Tabular. La elección del conversor importa para los archivos grandes y antiguos.

En el otro sentido, Tabular puede proporcionar interpolación de tablas externas a programas de simulación. El puente `ETABLE` y su enlace con `libtabular2` pertenecen a esa integración: no convierten una `TABLE` nativa del deck en un archivo externo sin una declaración y una llamada adecuadas.

Fuentes: `tabular2/src/sceptre-tabular.1`, `tabular2/src/pos4.c`, `sceptre2novender/nuevo/bin/sceptre2simul` y documentación de Tabular.

13.14. Las tres generaciones del formato binario Dirac

Origen: dos generaciones heredadas y una ampliación del proyecto. Son tres disposiciones del archivo binario de resultados. No deben confundirse con tres análisis físicos ni con la salida de texto `.sim`. La fase 2 escribe el binario por la unidad asociada a `fort.7`; el lanzador administra nombres como `caso.dirac`.

Generación	Directorio	Procedencia y reconocimiento
1. POS antiguo	Arreglos separados de 51 nombres, tipos, longitudes, direcciones y reservas en el registro inicial.	Heredado de 00.317, 2012.061 y 2013-0.6; firma inicial `DIRECTOR`; fin `ZZZZZZZZ`.
2. POS4	Entradas consecutivas de 32 bytes en un único registro de 2048 bytes.	Heredado del árbol SC1/SD2 de partida; cabecera `DIRECTORY`, `recno=1`.
3. POS4 extensible	La cabecera indica `NDIR`; el directorio ocupa los registros 1 a `NDIR`.	Ampliación del proyecto, octubre de 2026. El formato anterior es el caso `NDIR=1`.

En las generaciones segunda y tercera, cada entrada ocupa:

Desplazamiento, desde cero	Longitud	Campo
0–11	12 bytes	Nombre del vector.
12–22	11 bytes	Reserva; fecha en la cabecera.
23	1 byte	Tipo.
24–27	4 bytes	Longitud en elementos.
28–31	4 bytes	Registro de comienzo de los datos.

En la primera entrada, de tipo 0, la longitud vale 2048 y el último campo significa número de registros del directorio. En las entradas de datos, `recno` sigue siendo una dirección absoluta: el desplazamiento de lectura es (`recno` − 1) 2048. El tipo 4 identifica números `REAL*4`; el tipo 9 permite texto, como `HEADING`; el tipo 1 indica un bloque libre y 255, interpretado como −1, marca el final libre del directorio.

Un directorio de un registro tiene 64 posiciones contando la cabecera. No equivale a 63 salidas físicas disponibles: también ocupan entradas el eje, los títulos y las marcas que correspondan. `outgrande.d` prueba la ampliación con 120 salidas; al crecer el directorio se desplazan los registros de datos y se actualizan sus direcciones.

Detalle cotejado con el escritor actual: `POST` añade al final de cada vector numérico el mínimo y el máximo, después de las muestras. Esos dos valores cuentan en la longitud almacenada, pero no son muestras. Algunas notas históricas de `TERCERDIRAC.md` los sitúan al principio; no se reproduce aquí esa indicación porque contradice la asignación de `POST`.

El lector actual de Tabular acepta ambos órdenes de bytes y limita `NDIR` a 4096 como comprobación defensiva. Directorio extensible no significa ausencia de todas las cotas del lector o del programa. El manual de usuario no debe equiparar estas generaciones con nombres de tipos lógicos de salida del contrato sin comprobar la implementación de ese contrato.

Fuentes: `sceptre2novender/TERCERDIRAC.md`, `pos4_dir.f90`, `pos4_interface.f90`, `post.f90` y `tabular2/src/pos4.c`.

13.15. Plan de modernización: objetivos, resultados y avance

Origen: historia y estado de los cambios del proyecto. La base del trabajo es el SC1/SD2 26.131 de Wolf-Rainer Novender, conservado como referencia en `cuasioriginal`. El árbol de desarrollo es `nuevo`. Esta síntesis distingue el objetivo inicial, el resultado registrado y lo que todavía falta; no presenta como modificaciones del proyecto las capacidades que ya traía la base.

El criterio inicial fue cambiar de a un paso, conservar la referencia y comparar resultados del corpus contra patrones. Los cambios de forma deben mantener el comportamiento numérico; una reparación o ampliación se documenta por separado, porque puede cambiar deliberadamente un resultado antes incorrecto. El plan también recupera trabajo previo de otros árboles de SCEPTRE del proyecto, en lugar de atribuirlo todo a esta etapa.

13.15.1. Estado de los pasos del plan

Paso	Objetivo inicial	Resultado y alcance actual
0	Compiladores Fortran modernos.	Modernización de construcciones obsoletas concretada para el modo GNU utilizado. No equivale a certificar conformidad estricta con Fortran 2008/2018: quedan tipos y representaciones específicos que requieren tratamiento propio.
1	Texto junto al binario.	Concretado: la fase 2 produce texto nativo y Dirac. El lanzador conserva nombres como `caso.simnativo` y `caso.dirac`.
2	Entrada interactiva moderna en SD2.	Ya estaba resuelto en la base 26.131 mediante GNU Readline e historial. Se incorporó su detección a la configuración; no se atribuye al proyecto la sustitución que ya existía.
3	Unificar y ampliar el lanzador.	Opciones de entrada, bibliotecas, directorio de salida, configuración y control de ejecución incorporadas. El diseño evolucionó: no deben copiarse como opciones vigentes todas las propuestas iniciales. La escritura opcional del Dirac antiguo no quedó acreditada como implementada.
4	Construcción GNU real.	Concretado: Autoconf y Automake compilan las fases y SD2, con pruebas de corpus y objetivos de distribución e instalación. Libtool se añadió después para la biblioteca compartida.
5	Eliminar COMMON, ampliar memoria y preparar reentrancia y BLAS/LAPACK.	Conversión a módulos y memoria dinámica ampliamente concretadas; eliminación de la dependencia de `-fno-automatic`; estado de fase 2 intercambiable por circuito. La reentrancia completa, los límites restantes y el backend BLAS/LAPACK no deben declararse terminados.
6	Recuperar las mejoras útiles de otros árboles.	Lanzadores reubicables, configuración de usuario, mapas de código y empaquetado mediante SummaPack incorporados. La integración de ngp/Gnuplot se descartó expresamente: no es una deuda de implementación. El paquete compila durante su construcción, no en `postinst`.
7	Documentar.	En curso: plan, registros, contratos, documentación interna, traducción española y agregados independientes. Falta cerrar de manera sistemática la documentación de usuario y de órdenes prevista originalmente.
8	Lector alternativo slex/-bison.	Pendiente en este árbol. La fase 1 sigue leyendo el deck con el analizador Fortran. La existencia de slex en otros componentes del proyecto no demuestra su integración aquí.
9	Circuitos como módulos dinámicos y ejecución persistente.	Circuitos compilados a `.so`, validación de huellas, reutilización y `sceptreDU --servidor` concretados. El servidor actual ejecuta cada trabajo en un hijo y espera su resultado. No es todavía concurrencia de varios circuitos en un único proceso sin hijos; la sustitución de expresiones por ExprTk tampoco se deduce de la carga dinámica.
10	Eliminar ENTRY y GOTO.	Concretado en las fuentes de las dos fases. El Fortran emitido por la fase 1 constituye otro trabajo: su eliminación de saltos sigue parcial según el último registro documental.

13.15.2. Hitos y ampliaciones posteriores

12 de octubre: construcción GNU y primeras integraciones de lanzadores y empaquetado; recuperación del modo de convolución y sus pruebas.

14 de octubre: eliminación documentada de los 78 bloques COMMON; las fuentes pasan a módulos y se transforma también la interfaz con el programa generado.

19 de octubre: carga dinámica del circuito, huellas y reutilización; directorio Dirac extensible y adaptación de sus lectores; biblioteca compartida mediante Libtool.

20 de octubre: ampliaciones de memoria y de índices superiores a 999; incorporación del modificador `TOTAL`. Este último sí cambia el lenguaje de entrada, a diferencia de las reestructuraciones internas.

21 de octubre: retirada de `-fno-automatic`, estado intercambiable por circuito, servidor persistente y protección por falta de avance.

22 de octubre: eliminación de GOTO en las fuentes de las fases, correcciones de persistencia de variables y contraste adicional con el corpus de Siru. Continúa la revisión del Fortran generado y de la documentación.

Estas fechas son hitos de los registros, no una afirmación de que cada frente se inició y terminó en ese día.

13.15.3. Qué permite afirmar el avance registrado

La revisión estática de los archivos `.f90` realizada para esta síntesis encontró 265 fuentes en `sceptreM`, 177 en `sceptreDU` y 8 en `sd2`. Excluyendo comentarios y literales, no encontró instrucciones COMMON, ENTRY ni GOTO. Este dato corresponde al código fuente inspeccionado; no incluye instrucciones emitidas como cadenas de texto en el Fortran de los circuitos.

El registro `el manual` informa una comprobación de 97 casos del corpus en ambos modos de ejecución y una revisión adicional de 272 decks del corpus de Siru, de los cuales corrieron 240. Son mediciones históricas registradas, no pruebas repetidas al escribir este agregado. Su cantidad puede aumentar, y un patrón idéntico acredita una regresión concreta, no la ausencia de todos los defectos.

No se asigna un porcentaje único de terminación: tener carga dinámica, retirar un salto y cambiar la formulación de una matriz densa son trabajos de tamaños distintos. El avance se informa mediante resultados verificables y pendientes definidos.

13.15.4. Pendientes que siguen teniendo entidad propia

1. Evitar programas generados desmesurados al expandir simbólicamente ciertas matrices densas. El caso `resescalera1200.d` motivó un registro de aproximadamente 3,2 GB de Fortran generado; es un problema de formulación, no sólo de reservar más memoria.

2. Completar la modernización del Fortran generado y revisar las cotas que permanecen en caminos particulares.

3. Implementar y validar el backend BLAS/LAPACK. Las opciones de configuración ya declaradas no constituyen una implementación.

4. Integrar el lector slex/bison con pruebas de equivalencia y diferencias de sintaxis explícitas.

5. Si se requiere ejecución sin hijos, completar el aislamiento de unidades de archivo, estado del generado y tratamiento de fallos. El intercambio de estado y el servidor con `fork` son avances previos, no la solución completa de ese objetivo.

6. Actualizar de manera continua documentación y pruebas: los errores de persistencia encontrados por el corpus de Siru mostraron que el corpus local, por sí solo, no recorría todos los caminos relevantes.

Fuentes del balance: el agregado del plan de modernización; este manual; registros `PASO5`, `MEMORIA`, `SO`, `SERVIDOR`, `INSTANCIA`, `ESTRUCTURA`, `AUTOTOOLS` y `DEFECTOS`; configuración y fuentes de `nuevo`. Los textos conservan estados antiguos: una anotación posterior que resuelve o corrige un punto prevalece sobre su descripción histórica, sin borrar la historia.

13.16. Reconstrucción de funciones auxiliares de 32 bits

Las fuentes reconstruidas de `FMOS` y las rutinas de `c10` sustituyen objetos antiguos de 32 bits incompatibles con el enlace de 64 bits. Es una reparación de disponibilidad de funciones heredadas, no una nueva instrucción del lenguaje.

Las reconstrucciones están en `usrlib/c10.f90` y `usrlib/fmos.f90`. Las llamadas, los argumentos y las escrituras propias de las funciones heredadas se explican en `masexplicaciones/20-funciones-auxiliares-corpus.tex`.
Fuentes: `corpus/corpus/novender/c11.d`, `corpus/corpus/novender/c11.f90`, `corpus/corpus/novender/c12.f90`; `sceptre2novender/usrlib/README.md`.

13.17. Alcance de TOTAL frente a las inductancias

No se debe atribuir a la palabra `TOTAL` de los capacitores una extensión a las bobinas: ese cambio local tiene las restricciones descritas en los agregados 1 y 2.

La explicación física de las capacidades e inductancias variables heredadas se encuentra en `masexplicaciones/22-inductancias-y-capacidades-variables.tex`.

13.18. Archivos .d del corpus que incluyen extensiones de Siru

Origen: extensiones de otras herramientas del proyecto; no son ampliaciones del lector de SCEPTRE de Novender. El sufijo `.d` no prueba que todo el contenido pertenezca a ese lenguaje. El corpus general reúne también entradas híbridas, pruebas de Siru, fragmentos y archivos auxiliares.

Construcción encontrada	Interpretación que debe documentarse
`#include i01-externas.d`	En los casos preparados, SCEPTRE lo ignora como comentario; Siru incorpora el fragmento de declaraciones. No es inclusión nativa de SCEPTRE.
`EXTERNAL FMOS (5) ruta`	Declaración de Siru/Calcula para compilar y cargar una función. SCEPTRE necesita la función enlazada y no acepta esa declaración bajo `FUNCTIONS`.

`GLM=[21]`, `PGLM`	Selección/configuración del integrador de Siru; no equivale a una opción de `INTEGRATION ROUTINE` del manual de Novender.
`IMEX EXPLICITA`, `ROS`	Controles de otras familias de métodos en Siru. Su presencia en una prueba no acredita soporte en SCEPTRE.
`SPLIT`, `SPLIT A`, `SPLIT B`	Definición de subflujos; `SPLIT_H` pertenece a esas expresiones.
`SIRU ADAPTIVE STEP`	Control particular de Siru, distinto de los límites de paso clásicos.
`FCONVJ(H2)` con `H2(T)`	Variante de núcleo por función de Siru; no debe confundirse con los identificadores numéricos de núcleos del archivo auxiliar del SCEPTRE heredado.

Ejemplo híbrido deliberado:

```
#include i01-externas.d
MODEL DESCRIPTION
  ...
```

El fragmento separado declara:

```
FUNCTIONS
  EXTERNAL FMOS (5) ../../../sceptre2novender/usrlib/fmos.f90
```

Este arreglo permite aportar a Siru información adicional sin introducirla como instrucción activa de SCEPTRE. Requiere la configuración de funciones externas de Calcula. La ruta y el número de argumentos pertenecen a la declaración de Siru; la llamada `FMOS(...)` del circuito es la que comparte con SCEPTRE.

`corpus/corpus/inventados/caja-kj.d` contiene la variante funcional del núcleo y un control específico de Siru. No debe copiarse como ejemplo autocontenido de convolución nativa de Novender. Tampoco deben trasladarse las secciones `ELEMENT DESCRIPTION`, las condiciones con `TICK` o las llaves del grupo `SceptreNoPuede` a la gramática del manual por aparecer en un archivo con extensión `.d`.
Fuentes: `corpus/corpus/beispiele/i01-externas.d`, `corpus/corpus/beispiele/c10-externas.d`, directorios `corpus/corpus/GLM`, `corpus/corpus/IMEX`, `corpus/corpus/ROS`, `corpus/corpus/SPLIT`; `corpus/corpus/inventados/caja-kj.d`; `sceptre2novender/usrlib/README.md`.

13.19. Mapa de la ampliación documental con el corpus general

La revisión del árbol `corpus/` encontró 368 archivos con extensión `.d` el 22 de octubre de 2026. Se examinó su contenido textual y se cotejaron las construcciones con los capítulos traducidos, los agregados previos y las fuentes auxiliares pertinentes. No se ejecutaron de nuevo todas las simulaciones. La cantidad incluye copias del mismo material, fragmentos, bibliotecas, volcados de continuación y archivos vacíos; no representa 368 circuitos independientes válidos.

Tema y casos	Resultado del cotejo	Ubicación

`FC11`, `FC12`, `FMOS`, `FSAT`	La llamada externa está explicada; faltaban las dependencias concretas y las escrituras auxiliares.	Agregado 20.
`dos-cap-salto`, `dos-bobinas-salto`	La topología está explicada; faltaba advertir cómo se interpretan estas condiciones incompatibles y los comentarios trasladados de Siru.	Agregado 21.
`B07`, `B16`, `V050`	Ampliación práctica de las leyes diferenciales, las tablas de materiales y la comparación entre programas.	Agregado 22.
`GLM`, `IMEX`, `ROS`, `SPLIT`, núcleos por función	Son extensiones de Siru; no deben incorporarse como gramática de Novender.	Agregado 23.
`B03`, `C18`, `C19`	Faltaba separar procedencia SPICE, representación en SCEPTRE y disponibilidad en la biblioteca.	Agregado 24.
`DIODE Q`, tablas anidadas, `X EG15`	Aplicaciones de ecuaciones, abreviaturas y formato libre ya descritos; no se consideran nuevas funciones por aparecer en otros mazos.	Capítulo 2.
`FPULSE`, `FSINSQ`, fuentes nulas de medida	Ya están explicadas; los nuevos usos amplían ejemplos, no la gramática.	Capítulos 3 y 4.
AC, `COMPLEX`, matrices, barridos y `RERUN`	Ya documentados. La comparación con Siru no debe interpretarse como identidad de capacidades entre programas.	Capítulos 2, 3 y 5.
`ETABLE` en `con_FOR_1.d`	Es una dependencia externa del ejemplo, no una tabla definida por el mero nombre de la función. El registro consultado no obtuvo una serie de ese mazo.	Agregado 20; registros del corpus.

13.19.1. Criterios para ampliar los casos

1. Una construcción se presenta como heredada si pertenece al lenguaje o a las funciones recibidas de SCEPTRE; extraerla ahora de otro ejemplo no la convierte en desarrollo del proyecto.

2. Una reparación de enlace, una función reconstruida o una corrección del generador se distingue de una nueva instrucción.

3. Los informes antiguos describen ejecuciones fechadas. No se transfieren automáticamente al estado actual ni se presentan como nuevas verificaciones.

4. Los fragmentos `c10-externas.d` e `i01-externas.d`, los archivos `model.d`, los `end.d` y los volcados `datasave` se distinguen de circuitos completos.

5. Los originales extraídos de PDF pueden contener errores y omisiones. Un listado conservado por procedencia no debe publicarse como ejemplo validado sin cotejo adicional.

Las explicaciones del SCEPTRE de Novender se trasladaron a `masexplicaciones/` y se reúnen en un anexo del libro mediante `masexplicaciones/anexo.tex`. Las modificaciones del proyecto permanecen separadas en `agregados/`. Los ejemplos del corpus no se modificaron; los listados del anexo tienen sus propios archivos.
Fuentes de cobertura y procedencia: `corpus/README.md`, `corpus/SCEPTRE2.md`, `corpus/SCEPTRE-distintos.md`, `corpus/corpus/beispiele/README.md`, `corpus/corpus/novender/README.md`.

13.20. COMPUTER TIME LIMIT: texto heredado y comportamiento actual

Origen: control heredado con activación y comportamiento local documentados el 22 de octubre de 2026. Deben separarse la descripción del libro y la implementación examinada. La mera presencia de una instrucción en el lector no demuestra que su efecto esté implementado.

Referencia	Qué establece
Texto traducido, apartado 2.10	Presenta `COMPUTER TIME LIMIT` como límite de tiempo transcurrido en minutos, con finalización ordenada.
Lector actual	Asocia la orden a `XMXERT`, posición 3 del vector de controles.
`OUTP` actual, transitorio	Cuando transcurre el intervalo positivo `XMXERT`, medido en segundos con `SYSTEM_CLOCK`, solicita el guardado mediante `KWAY=9`.
Programa generado	El camino `KWAY=9` llama a `SAVEPD(1)`, vuelve a poner `KWAY=0` y continúa la integración.

Uso del control en el código actual:

```
RUN CONTROLS
  STOP TIME = 1000
  COMPUTER TIME LIMIT = 30
```

En una ejecución transitoria que dure lo suficiente, esto solicita guardados aproximadamente cada 30 segundos de reloj del ordenador. `STOP TIME=1000` sigue siendo el tiempo final del modelo, en las unidades coherentes elegidas para el circuito. No significa ejecutar durante 1000 segundos reales ni detenerse después de 30 segundos.

- La comprobación se hace al pasar por `OUTP`; no es una interrupción asíncrona capaz de detener una función de usuario bloqueada.
- El intervalo se cuenta desde el primer punto y se renueva tras cada solicitud de guardado. Con valor cero no se activa este guardado periódico.
- El estado de continuación se guarda en `datasave.dat`; no debe confundirse con el archivo de resultados Dirac ni con una exportación textual de variables.
- Este camino está dentro de la rama no AC. No debe prometerse que la orden produzca el mismo guardado periódico en un barrido de frecuencias.
- El texto heredado sobre minutos y finalización no describe este comportamiento actual. Los comentarios históricos que dicen que ningún mazo lo usa tampoco sustituyen la prueba posterior `guardado.d`.

`SOLUTION TIME LIMIT`, asociado a `XMXSLT`, es otro control: el libro lo presenta como límite de CPU. No es sinónimo de `COMPUTER TIME LIMIT`; la presencia del campo y su impresión en el listado no bastan para afirmar que se aplique un límite efectivo. El cambio aquí descrito implementa específicamente el guardado periódico con `XMXERT`.

La nota de desarrollo registra una prueba de `guardado.d` con intervalo muy pequeño, guardados repetidos y el mismo resultado `.sim` que `a01`; esa prueba no se volvió a ejecutar al redactar este agregado.

Fuentes: `sceptre2novender/nuevo/sceptreM/module_runcon.f90`, `sceptre2novender/nuevo/sceptreM/wrtout.f90`, `sceptre2novender/nuevo/sceptreDU/outp.f90`; `sceptre2novender/pasos/kway9-guardado.diff`; `sceptre2novender/corpus/guardado.d`.

13.21. El paquete binario y el cambio de sistema operativo

Origen: decisión del proyecto. SCEPTRE no es un programa que se instala y corre: la fase 1 *escribe* el programa Fortran del circuito y `sc1` lo compila y lo enlaza *en cada corrida*. De eso se siguen dos consecuencias que este agregado documenta.

13.21.1. Lo que el paquete necesita en cada corrida

Lo que para un paquete común son dependencias de construcción, acá son de ejecución. El programa generado se compila con `gfortran` y se enlaza con

```
-ltabular2 -llapack -lblas -lumfpack -lstdc++
```

contra la biblioteca de fase 2 y los `.mod` que instala el propio paquete. Por eso el paquete declara `gfortran`, `libtabular2-dev`, `liblapack-dev`, `libblas-dev` y `libsuitesparse-dev` en `Depends` y no sólo en `Build-Depends`: sin ellas la primera corrida muere en el enlazador con `cannot find -llapack`.

13.21.2. Por qué un paquete binario muere al cambiar el sistema

Un archivo `.mod` de gfortran lo lee *solamente* el gfortran que lo escribió; su formato tiene número de versión propio, que el compilador publica en el encabezado del archivo (`GFORTRAN module version '16'`) y que Debian expone como el paquete virtual `gfortran-mod-16`. Cuando la distribución pasa a un gfortran nuevo, los `.mod` instalados dejan de servir y el primer compilado del circuito falla con `Cannot read module file … created by a different version of GNU Fortran`. La biblioteca estática arrastra además el compilador y la `libc` con que se construyó.

Para que eso falle a tiempo, y no en medio de una simulación, el paquete declara `${fortran:Mod}` en `Depends`: `debian/rules` lee la versión de los `.mod` del gfortran con que construye y la convierte en `gfortran-mod-N`. Con un gfortran nuevo el paquete deja de ser instalable, que es el aviso correcto.

13.21.3. La capacidad Ajolote: reconstruirse desde las fuentes que trae el paquete

Todo paquete binario muere al cambiar el sistema operativo, y lo habitual es que el sistema nuevo traiga otro paquete construido para él. Pero nada asegura que éste se distribuya para cada sistema nuevo. Por eso, decisión del autor, **cada paquete binario se trae sus propias fuentes** y puede rehacerse a sí mismo.

A esa propiedad la llamamos **capacidad Ajolote**: el ajolote se reconstruye desde casi cualquier parte, siendo un animal complejo. La *Hydra* común lo hace más —y además no muere en condiciones favorables—, pero SCEPTRE es complejo, así que se parece más a un ajolote.

Lo que el paquete lleva adentro:

`share/sceptresimusol/fuentes/` el `tar` de las fuentes del propio paquete y su archivo `.summa`, que es lo que `sp` necesita para rearmarlo.

`share/sceptresimusol/gfortran-mod` la huella: la versión de los `.mod` con que se construyó.

Los manuales escaneados no viajan en ese `tar` (son 49 de sus 51 MB y no hacen falta para construir), con lo que las fuentes ocupan poco más de un megabyte dentro del paquete.

Cuando el compilado del circuito falla, `sc1` compara esa huella con la versión de `.mod` que escribe el gfortran de la máquina y, si no coinciden, explica la causa y da la orden exacta:

```
sc1: EL COMPILADOR DE ESTA MAQUINA NO PUEDE LEER LOS .mod DEL PAQUETE
sc1:   los .mod instalados son de gfortran-mod-16 y este gfortran
sc1:   escribe gfortran-mod-17
sc1:   el paquete trae sus fuentes en /usr/share/sceptresimusol/fuentes,
sc1:   asi que se rehace aca mismo:
sc1:     sp --para /usr/share/sceptresimusol/fuentes install sceptresimusol
```

La comprobación no cuesta nada en la corrida normal: se hace sólo cuando el compilado ya falló.

El paquete de *fuentes* no lleva nada de esto: no lo necesita.

13.21.4. El paralelo con DKMS

La capacidad Ajolote (13.21.3) tiene un pariente conocido. La forma es la misma que usa DKMS (*Dynamic Kernel Module Support*) para los controladores que no están en el núcleo oficial: el paquete no instala el módulo compilado, instala sus fuentes en `/usr/src/` y el sistema las recompila cuando cambia el núcleo. Las diferencias con lo que hace este proyecto son tres:

- DKMS sirve sólo para módulos del núcleo; esto es un programa de espacio de usuario.
- DKMS lo dispara el sistema al instalar un núcleo nuevo; acá avisa el propio `sc1` y la reconstrucción la ordena la persona.
- DKMS existe en las distribuciones de Linux y no fuera de ellas; la reconstrucción de acá la hace `sp` (summapack), que arma `.deb`, `.rpm` y `.tgz`.

13.22. Los archivos propios del usuario: `sceptre-user-setup`

Origen: infraestructura incorporada por el proyecto (paso 6 del plan). La instalación queda de sólo lectura: es de `root` y la comparten todos los usuarios de la máquina. Pero SCEPTRE necesita escribir y el usuario necesita cambiar cosas —la biblioteca de modelos, la de usuario, la configuración—, así que `sceptre-user-setup` le deja copias propias en su directorio y le arma la configuración que las apunta.

Se corre una vez, cuando se quiere, y no hace falta ser `root`:

```
sceptre-user-setup [DIR-DATOS] [DIR-CONFIG] [DIR-EJEMPLOS]
```

13.22.1. Qué hace, en orden

1. **Crea los tres directorios.** Por omisión `~/.local/share/sceptre/lib`, `~/.config/sceptre` y `~/sceptre-ejemplos`; respeta `XDG_DATA_HOME` y `XDG_CONFIG_HOME`, y los tres se pueden dar como argumentos.

2. **Crea el directorio de modelos propios**, `DIR-CONFIG/modelos`, con su `LEEME`: ahí van los modelos del usuario, y `sc1` los manda en el texto del deck junto con los de la biblioteca instalada, que ya no hay que copiar (véase 13.24). **Ya no copia ninguna biblioteca.** La de modelos no, y ésa era justamente la trampa: `sceptre.conf` apuntaba `MODELLIB` a la copia, así que los modelos nuevos de una versión del paquete no llegaban nunca. Ahora la instalada se usa donde está y lo propio va en el directorio de modelos. `DIR-DATOS/lib` se sigue creando: ahí deja `sc1` la biblioteca dinámica de las funciones propias. **Ya no copia** `usrlib.a` (5 de noviembre de 2026): el paquete no la instala más —sus funciones son fuentes del proyecto, compiladas con la fase 2— y antes el *setup* la buscaba y *cortaba* si no estaba, así que sobre un paquete instalado fallaba antes de hacer nada.

3. **Copia la configuración de** `sd2` (`sdrc`) a `DIR-CONFIG`.

4. **Escribe** `DIR-CONFIG/sceptre.conf`, que es lo que lee `sc1`, con `OUTDIR=.` y, comentados, `MODELLIB` —para quien quiera *reemplazar* la biblioteca instalada en lugar de sumarle sus modelos—, `USRLIB` —para quien tenga su propia biblioteca estática— y `FC` y `FFLAGS`, por si quiere cambiar el compilador o sus opciones. **Si ese archivo ya existe no lo toca** y lo dice.

5. **Copia los ejemplos** (`samples/*.d`) a `DIR-EJEMPLOS`.

6. **Pone los lanzadores en el escritorio** (desde el 5 de noviembre de 2026). El paquete instala tres entradas `.desktop` en `share/applications` —de ahí las toma el menú de aplicaciones solo— y un icono en `share/icons/hicolor/scalable/apps`; lo que hace falta por usuario es ponerlas en *su* escritorio, y eso es este paso. Las tres son: **SCEPTRE (sd2)**, que abre el intérprete en una terminal; **Manual de SCEPTRE**, que abre el PDF instalado; y **Ejemplos de SCEPTRE**, que abre la carpeta de ejemplos (con la ruta real, si se pidió otra). El directorio del escritorio lo dice `xdg-user-dir`; si no está, se prueban `~/Escritorio` y `~/Desktop`. En GNOME un lanzador en el escritorio tiene que estar marcado como de confianza, y eso lo hace `gio set … metadata::trusted`; donde no hay `gio` no es un error.

7. **Termina listando** dónde quedó cada cosa, incluidos cuántos iconos puso y en qué directorio, con la prueba: `cd ~/sceptre-ejemplos && sc1 a01`.

13.22.2. Qué instala el paquete, y dónde

Los manuales: en `/usr/share/doc/simusol/sceptresimusol/`, no en el `doc/` del paquete. Es la convención de los árboles `siru` y `calcula` (decisión del autor, 5 de noviembre de 2026): **todos** sus programas ponen el manual bajo `/usr/share/doc/simusol/`, en un subdirectorio por programa —como ya hacía `dia-simusol` con su `manual.pdf`—, así la documentación de Simusol queda junta y no repartida en un directorio por paquete.

Lo propio del paquete: `LEEME`, `COPYING` y `00README.1st` siguen en `/usr/share/doc/sceptresimusol/`.

Las páginas de manual: `sc1`, `sd2`, `sceptre`, `sceptre2simul` y `sceptre-user-setup` en `man1`.

La biblioteca de modelos: `modellib.dat`, donde la encuentra la rutina reubicable. **No** se instala `usrlib.a`: desde el 5 de noviembre de 2026 no existe más como pieza del paquete, y su directorio `usrlib/` salió del árbol (lo que traía está en `extras/`, y los objetos originales en `extras/objetos/`).

Los lanzadores y el icono: `share/applications` y `share/icons/hicolor/scalable/apps`. Antes el paquete no instalaba ninguno: el del SCEPTRE viejo lo hacía su `postinst` con un `gnome_icon_install` que ya no existe.

Busca lo instalado con la misma rutina reubicable que `sc1` (`lib/`, `share/sceptresimusol/`, `etc/`), así funciona igual desde el árbol de compilación o desde el paquete instalado. Ya no necesita encontrar ninguna biblioteca para hacer su trabajo.

13.22.3. Para qué sirve `modellib.dat`

Desde el 5 de noviembre de 2026 el usuario no necesita copiarla: sus modelos van en su directorio de modelos y `sc1` los junta con los de ésta (véase 13.24). Lo que sigue es qué es y por qué `sc1` la mira. Es la **biblioteca de modelos permanentes**: un archivo de texto con un directorio de nombres al principio y, debajo, cada modelo guardado —`MODEL FD300 (PERM) (1-3)` y sus `ELEMENTS`—. Trae 22: diodos (`1N914`, `1N4001`, `POWERDIODE`), transistores (`2N1711`, `2N2905`, `2N706A`), tiristores (`SCR`, `THYRISTOR1`, `SPICE_SCR`, `C358E`), un operacional (`MC709C`)…

Cuando un deck nombra un modelo en lugar de describir el subcircuito, **la fase 1 lo saca de acá** y lo expande en el circuito. O sea que es *dato*, no código: a diferencia de `usrlib.a` no se compila ni se enlaza, se lee en cada corrida. Por eso tiene sentido que el usuario tenga su copia —agregar un modelo propio o corregir uno no debería pedir permiso de `root`— y por eso `sc1` la mete en la huella del caso (`HM`): si cambia la biblioteca, el circuito se rehace, porque los modelos que expandió ya no son los mismos. La elige `-m`, o `MODELLIB` en `sceptre.conf`.

13.22.4. Quién manda sobre quién

`sc1` lee `sceptre.conf`, pero las opciones de la línea de comando (`-m` para la biblioteca de modelos, `-u` para la de usuario, `-o` para el directorio de salida) le ganan a lo que diga el archivo, y `SCEPTRE_NO_CONF=1` lo saltea del todo. El orden es el habitual: lo que se pide ahora pesa más que lo configurado, y lo configurado más que lo instalado.

13.22.5. Qué reemplaza

El `postinst` del Sceptre-Simusol de 2017 hacía esto en la instalación y *para cada usuario de `/home`*: recorría los usuarios y les corría `sceptre-simusol-install`. Ahora la instalación no toca ningún directorio personal: cada usuario corre `sceptre-user-setup` cuando le hace falta. Es la diferencia entre un instalador que decide por todos y uno que deja decidir, y además evita que una instalación escriba en `~` de gente que quizá no usa SCEPTRE.

Lo que no hace: no instala nada en el sistema, no pide `root`, no borra lo que el usuario ya tenía y no vuelve a copiar la configuración si ya está.

13.23. Las funciones propias del usuario, como módulo dinámico

Origen: decisión del proyecto. El manual original ya admite que una instrucción aritmética llame a «*external functions*»: las de la biblioteca FORTRAN y *subprogramas propios*. Lo que faltaba era cómo los agrega quien no compila SCEPTRE: hasta ahora había que rearmar la biblioteca `usrlib.a` o pasarla con `-u`, o sea tocar la instalación.

Ahora hay un lugar y un método: el usuario escribe sus funciones en un directorio propio y `sc1` **arma solo la biblioteca dinámica y la usa**.

13.23.1. El lugar

`sceptre-user-setup` crea `~/.config/sceptre/funciones/` con un `LEEME` adentro que explica todo esto. Ahí van los `.f90`, uno o varios. El directorio se puede cambiar con `SCEPTRE_FUNCIONES`.

13.23.2. El método

1. El usuario escribe la función: Fortran de toda la vida, `real*8`, argumentos por referencia, nombre de hasta seis caracteres.
2. La llama desde el deck **dentro de una instrucción aritmética**, que es la forma que el manual documenta para las funciones externas, no como valor directo de un elemento:

```
ELEMENTS
  E1, 1-0 = EQUATION 1 (2.5, TIME)
  R1, 1-2 = 10.
  C1, 2-0 = 1.E-3
FUNCTIONS
  EQUATION 1 (A,B) = (MIRAMP(A,B))
```

3. Corre `sc1` como siempre. Si algún `.f90` es más nuevo que la biblioteca, `sc1` la rehace —`funciones-usuario.so`, con `-fPIC -shared`, en `~/.local/share/sceptre/lib/`— y la enlaza, en el modo estático y en el dinámico. Si no hay ningún `.f90`, nada cambia.

Si las funciones llaman una biblioteca C compartida, `SCEPTRE_FUNCIONES_LIBS` agrega sus opciones de enlace al construir `funciones-usuario.so`; por ejemplo, `-lradia`. El puente de radiación solar está en `extras/radia/` y el ejemplo de un día en `corpus/radia/mediodia.d`. Requiere que `libradia` esté instalada o que se indiquen su ruta de enlace y su ruta de carga.

13.23.3. Cómo se escribe una función

Una función de usuario es Fortran de toda la vida. Cuatro reglas, y las cuatro importan:

Es una `function` de `real*8` y sus argumentos son `real*8`. SCEPTRE la llama por referencia, como llama a las de su propia biblioteca; no hay que declarar nada del lado del deck.

El nombre, hasta seis caracteres. Es el límite de los nombres del lenguaje de SCEPTRE, no del Fortran: un nombre más largo no se puede nombrar desde el deck.

Nada de estado entre llamadas si se puede evitar. La fase 2 la llama en cada paso de integración, y en un paso que después se rechaza, así que una función que recuerda lo anterior puede dar resultados que dependen del tamaño de paso. Si hace falta memoria, que sea función del tiempo que recibe, no de cuántas veces la llamaron.

Que siempre vuelva. Una función que se cuelga no cuelga la corrida para siempre: la corta el latido (capítulo 6), pero la corrida se pierde.

Un ejemplo completo, el de la prueba. En `~/.config/sceptre/funciones/mis-funciones.f90`:

```
real*8 function miramp (x, pendiente)
implicit none
real*8, intent(in) :: x, pendiente
   if (x .gt. 0.d0) then
      miramp = pendiente * x
   else
      miramp = 0.d0
   end if
end function miramp
```

En un archivo puede haber varias, y puede haber varios archivos: `sc1` los compila todos juntos en `funciones-usuario.so`.

13.23.4. Cómo se la llama desde el deck

No se pone la función como valor de un elemento: va *adentro* de una instrucción aritmética —`EQUATION` o `EXPRESSION`—, que es la forma que el manual original documenta para las «external functions»:

```
ELEMENTS
  E1, 1-0 = EQUATION 1 (2.5, TIME)
  R1, 1-2 = 10.
  C1, 2-0 = 1.E-3
FUNCTIONS
  EQUATION 1 (A,B) = (MIRAMP(A,B))
```

Puesta directamente como valor de un elemento —`E1, 1-0 = MIRAMP(2.5,TIME)`— el que se queja es el lector de tarjetas: `BCDFP`, error 46, `INVALID CONSTANT SPECIFIED`. Y ojo con dos cosas del lenguaje: las constantes numéricas llevan punto decimal (`2.5`, `10.`, no `10`) y los nombres van hasta seis caracteres.

13.23.5. Y si está escrita en C

Hoy el directorio toma `.f90`: eso es lo que `sc1` compila. Una función en C se puede usar, pero por el camino del proyecto, no por éste: se escribe un envoltorio Fortran con `iso_c_binding` (`bind(c)`) y se compila con la fase 2, que es como entran `calcula_tuberia.c` y el puente de ExprTk en `extras/` (véase `extras/README.md`).

13.23.6. Dos cuidados que el mecanismo se toma

La huella: el `.so` entra en la huella del caso. Cambiar una función obliga a rehacer el circuito, así que no queda una corrida con la versión vieja de la función —que es el error peligroso, porque no avisa.

No pagar lo que no se usa: la biblioteca se rehace sólo si alguna fuente es más nueva, y `--sin-funciones` la saltea por esta vez.

13.23.7. Lo que lo comprueba

`tests/prueba-funciones.sh` corre un deck que llama a una función que **sólo** existe en la biblioteca del usuario, y verifica cinco cosas: que `sc1` arme la biblioteca; que el modo estático dé el número exacto que la cuenta predice (una rampa de pendiente k por un RC de constante τ da $V_C = k(t-\tau)$: con $k = 2{,}5$, $\tau = 0{,}01$ y $t = 0{,}1$, eso es 0,2275, y el deck da 0,22775); que el modo dinámico dé lo mismo; que al cambiar la función cambie la corrida; y que con `--sin-funciones` el deck *no* corra, lo que prueba que la función venía de ahí y no de otro lado.

13.23.8. Qué pasó con `usrlib.a`

Desaparece como vehículo de las funciones del proyecto: las que traía —`FMOS`, `FTABL1`, `FSAT`, `FC11`, las de SPICE…— no eran del usuario sino del proyecto, y hoy están en `extras/`, compiladas con la fase 2 (véase 13.22 y `extras/objetos/LEEME.md`). La opción `-u` sigue existiendo para quien tenga su propia biblioteca estática, y es opcional: si no hay ninguna, no se enlaza nada. El directorio `usrlib/` salió del árbol, y `sceptre-user-setup` dejó de copiar la biblioteca: la única que le copia al usuario es la de modelos (véase 13.22).

13.23.9. Ejemplo desde Dia

Abra `con_FOR.s.dia` del catálogo de Dia-Simusol. El diagrama selecciona `MOTOR = SCEPTRE` y combina dos orígenes de código Fortran. La definición de `F1()` se edita en el propio dibujo. Para `FFFF` el lanzador recupera la función de `funciones.for` y prepara su compilación para la corrida.

13.24. Los modelos propios del usuario, y cómo entran

Origen: decisión del proyecto. La biblioteca de modelos era *un* archivo que la fase 1 abría por nombre: `modellib.dat`, en el directorio de trabajo, con un índice al principio. Como era uno solo, había que elegir: o la instalada, o la copia del usuario. El que agregaba un modelo propio se quedaba sin los que trajera una actualización del paquete, y el que quería los nuevos perdía los suyos.

Ahora hay **dos lugares** —la instalada, en `share`, que no se toca, y la del usuario, en su directorio de modelos— y **es `sc1` el que resuelve**: mira qué modelos usa el deck y no define, los busca primero entre los del usuario y después en la instalada, y los manda **en el texto del deck**, como una sección de modelos, que es como un deck trae sus modelos desde siempre. La fase 1 no se entera de que había dos archivos; no tiene por qué saberlo.

13.24.1. El lugar

`sceptre-user-setup` crea `~/.config/sceptre/modelos/` con un `LEEME` adentro. Ahí van los `.dat`, uno o varios, y cada uno puede traer varios modelos. El directorio se elige con `SCEPTRE_MODELOS_DIR`.

Un archivo del usuario se puede escribir de dos maneras, y las dos se leen: con el índice del formato de `modellib.dat`, o —lo normal— con sólo los `MODEL`, que es lo mismo que iría en la `MODEL DESCRIPTION` de un deck:

```
MODEL MIDIODO (PERM) (A-K)
ELEMENTS
  JD, A-K = X1 ( MAX(0.,VJD**5) )
  CD, A-K = 1.E-9
```

13.24.2. Cómo se escribe un modelo

Un modelo es un subcircuito con nombre y terminales. Se escribe igual que el circuito, y de hecho es lo mismo que iría en la `MODEL DESCRIPTION` de un deck; lo único propio es el renglón de encabezado:

```
MODEL NOMBRE (PERM) (TERMINAL-TERMINAL-...)
```

Los terminales son los nodos por donde el modelo se conecta, y se nombran como se quiera: letras (`A-K` de un diodo, `B-E-C` de un transistor, `B1-E-B2` de un unijuntura, `G-A-K` de un tiristor), números (`1-3`) o palabras (`INN-INP-GND-OUT` del operacional `MC709C`). Adentro, **todo nodo que no sea terminal es interno**, y no se ve desde el circuito: en el `2N706A` los nodos `1` y `2` son internos y `B`, `E`, `C` son los terminales.

Debajo del encabezado van las mismas secciones que un circuito, y en este orden: `ELEMENTS`, `DEFINED PARAMETERS`, `FUNCTIONS` y, si se quiere, `OUTPUTS`. Un ejemplo completo, el `2N706A` de la biblioteca:

```
MODEL 2N706A (PERM) (B-E-C)
  ELEMENTS
    CE, 1-E = Q1(5.,70.,J1)
    CC, 1-2 = Q1(8.,370.,J2)
    RB, B-1 = .3
    RC, C-2 = .015
    J1, 1-E = DIODE EQUATION(1.E-7,35.)
    J2, 1-2 = DIODE EQUATION(5.E-7,37.)
    JA, E-1 = .1*J2
    JB, 2-1 = P1*J1
    JX, 2-1 = 0
  DEFINED PARAMETERS
    P1 = 0.98
  FUNCTIONS
    Q1(AA,BB,CC) = (AA+BB*CC)
```

Ahí está casi todo lo que se puede usar: elementos de valor constante (`RB`), de valor dado por una instrucción aritmética que depende de una corriente del propio modelo (`CE`), fuentes dependientes de otro elemento (`JA`, `JB`), un diodo por su ecuación (`J1`) —o por una tabla, `DIODE TABLE 1`—, un parámetro definido que se usa como valor (`P1`) y una función del modelo (`Q1`). La primera letra del nombre dice el tipo del elemento, como en el circuito: `R` resistencia, `C` capacitor, `L` inductancia, `E` fuente de tensión, `J` fuente de corriente.

13.24.3. Cómo se usa desde el circuito

Como un elemento más, dando los nodos del circuito en el **orden de los terminales** del modelo:

```
ELEMENTS
  T1, 3-4-5 = MODEL 2N706A
  T2, 4-8-7 = MODEL 2N706A
```

`T1` conecta su `B` al nodo 3, su `E` al 4 y su `C` al 5. El mismo modelo se puede instanciar tantas veces como se quiera, y cada instancia es independiente.

Al expandirse, cada nombre de adentro del modelo pasa a llamarse **nombre más instancia**: los elementos de `T1` son `CET1`, `CCT1`, `RBT1`, `J1T1`… y su parámetro, `P1T1`. Así se los pide en `OUTPUTS` (`VCET1`, `J1T1`) y así aparecen en el listado. De paso, eso acota los nombres: nombre de adentro más nombre de la instancia tienen que seguir cabiendo en seis caracteres.

Entre paréntesis se le puede pedir dos cosas a la instancia:

```
T1, 3-4-5 = MODEL 2N706A (PRINT)
T2, 6-1-8 = MODEL 2N706A (TEMP,CHANGE P1=.975)
```

`(PRINT)` hace que el listado muestre el modelo expandido, que es la manera de ver qué entró de verdad; `CHANGE` le cambia un parámetro definido *en esa instancia* sin tocar el modelo. En el árbol, `corpus/modcambios.M.d` usa justamente eso.

13.24.4. Quién gana

El deck. Un modelo que el deck define en su propia `MODEL DESCRIPTION` no se manda: manda el del deck.

Después, el del usuario. Si el nombre existe en las dos, gana el del usuario, y el listado lo dice. Es la manera de corregir un modelo que viene con el paquete sin tocar la instalación.

Y si no, el instalado. Que sigue llegando entero, con los modelos nuevos que traiga cada versión.

Un modelo del usuario puede usar otro modelo: `sc1` va a buscarlo también, y así hasta cerrar.

13.24.5. Qué pasa al pasarlos

`(PERM)` pasa a `(TEMP)`. En un deck, `(PERM)` quiere decir «guardalo en la biblioteca permanente», y si el nombre ya está da el error 39, `REDUNDANT MODEL HAS BEEN SPECIFIED`; `(TEMP)` es un modelo de esta corrida, que es justo lo que hace falta. Así que da igual cómo esté escrito el archivo.

13.24.6. Lo que queda en el listado

Un renglón por modelo, antes de todo lo demás:

```
sc1: modelo 2N706A <- /home/juan/.config/sceptre/modelos/mio.dat
sc1: el modelo 2N706A tambien esta en
     /usr/local/share/sceptresimusol/modellib.dat: gana el de
     /home/juan/.config/sceptre/modelos/mio.dat
```

Y si el deck pide un modelo que no está en ninguna parte, lo dice ahí; el error, como siempre, lo da la fase 1.

13.24.7. Lo que lo comprueba

`tests/prueba-modelos.sh`, cinco cosas: que un deck que usa un modelo de la biblioteca instalada corra y que el listado diga de qué archivo salió; que `--sin-modelos` ignore el modelo propio y vuelva al instalado; que cambiar el modelo propio **rehaga** el circuito corriendo dos veces en el mismo directorio —ahí se ve si la huella lo mira o si se reutiliza el `.so` viejo—; que un modelo propio de igual nombre gane y cambie la corrida; que uno con nombre nuevo se use y que sin el mecanismo el mismo deck *no* corra; y que el archivo del usuario se pueda escribir con índice o sin él.

La equivalencia se midió sobre `samples/a08.d` con su modelo sacado del deck: por la biblioteca y por el texto, el `datasave`, el `plotsave`, el `acsave` y el listado numérico salen **idénticos**; sólo cambian la hora y las huellas. Con `(PERM)` tal cual, en cambio, el error 39, que es lo que obliga al cambio a `(TEMP)`.

13.24.8. La fase 1 ya no abre ninguna biblioteca

Mientras el mecanismo nuevo convivió con el viejo, la fase 1 seguía abriendo `modellib.dat` por nombre, en el directorio de trabajo, y `sc1` le ponía un *symlink*. Eso se fue: la unidad se abre vacía, como `scratch` (`sceptreM.f90:200`). No hay archivo que pueda faltar, así que **se fueron también los códigos 105 y 103** —«falta `modellib.dat`» y «faltaba pero la corrida siguió con uno vacío»— y de `sc1`, los tres *symlinks* (la corrida normal, cada `fase1-MODELO` y el modo `-lote`).

La unidad sigue existiendo porque es la biblioteca *permanente* de `MDEDIT`, y ahí está lo único que se pierde, que conviene decir claro.

13.24.9. Editar la biblioteca desde un deck: suspendido

SCEPTRE sabía **escribir** su biblioteca, y lo ordenaba el deck, no el usuario desde afuera: un modelo que en la `MODEL DESCRIPTION` llevaba `(PERM)` —o `PERMANENT`, o `MASTER`— no era un modelo de la corrida sino una *entrada nueva en la biblioteca permanente* (`mdedit.f90:206--248`: `INDSRH` lo busca, el índice crece), con `(DELETE)` se borraba una, y al terminar `WRTTPE` (`mdedit.f90:266`) reescribía la biblioteca entera. Si el nombre ya estaba, el error 39, `REDUNDANT MODEL HAS BEEN SPECIFIED`. El idioma de siempre para reemplazar un modelo del paquete eran las dos tarjetas juntas, borrar y volver a guardar, como en `samples/a03.d`.

Así que a la pregunta de quién podía ordenarlo la respuesta era: **cualquiera que escribiera el deck**; y lo que se modificaba era el archivo que `sc1` le pasara. Eso nunca fue una manera confiable de guardar un modelo, por dos razones: durante años ese archivo fue la *copia* del usuario —con lo que un modelo nuevo se iba a un archivo que después tapaba a la biblioteca instalada, y las versiones del paquete dejaban de llegar— y cuando no, era el de la instalación, de `root`, donde un usuario no puede escribir. El resultado dependía de los permisos.

Esa función queda suspendida (decisión del autor, 5 de noviembre de 2026): **no se ejecuta**. La fase 1 avisa en el listado y sigue:

```
MDEDIT  : editar la biblioteca esta SUSPENDIDO: el modelo 2N706A pidio
          (PERM) y vale solo para esta corrida.
MDEDIT  : un modelo que tiene que quedar va en el directorio de modelos
          del usuario.
```

Con `(PERM)` o `MASTER` el modelo vale para *esta* corrida, como si dijera `(TEMP)`, que es lo que el deck quiere casi siempre.

La tarjeta `(DELETE)` se ignora, y se avisa.

Pedir la biblioteca permanente —el `(PERM)` en el instanciado, el `MASTER` como cinta de modelos, el barrido alternativo del aviso 67— va a la única cinta que queda, la de esta corrida (`cdprs.f90`, `modprs.f90`, `mmodif.f90`).

Un deck que usaba todo eso sigue corriendo y da lo mismo: `samples/a03.d`, que borra el `2N706A`, lo vuelve a guardar con `(PERM)` y después lo instancia con `(PERM)`, corre con código 0.

El lugar de un modelo que tiene que quedar es el directorio de modelos del usuario: es un archivo suyo, lo edita cuando quiere, sobrevive a la actualización del paquete, le gana al de la biblioteca instalada y el listado lo nombra en cada corrida. Lo que se pierde con la suspensión es poder guardar un modelo *desde un deck*, que es precisamente lo que no funcionaba.

13.24.10. Cómo se leía, para el que quiera seguirlo

El formato lo fijaba la fase 1 (`sceptreM/mdedit.f90:183`): el primer registro, `(18x,G12.4)`, la cantidad de modelos; después un registro `(a18,g12.4)` por modelo con su nombre y su cantidad de tarjetas; y después los cuerpos, en ese orden, que se copiaban tal cual a la biblioteca temporal. Aparte, `common_cotas.f90` hacía una pasada barata sobre el mismo índice para dimensionar. `sc1` sigue leyendo ese formato —es el de la biblioteca instalada, y un archivo del usuario puede venir así— y es contra esa lectura que se comparó la equivalencia.

13.25. La lectura por slex y el .dsa directo

Origen: dos pedidos del autor. El primero (05/11/2026): «la idea es que se deje de usar ese análisis en fortran […] nada de texto debe ser analizado por fortran», y «no debe reemplazarse lo viejo sino tiene que ser alternativo con una opción». El segundo (10/11/2026): «debe ir de dsa directo a las variables fortran, y de .d si lo dejamos también. O sea: llega un `.d`, slex lo manda a Fortran; llega un `.dsa`, otro parser de slex lo lleva a Fortran»; y sobre lo que había entonces —correr un `.dsa` traduciéndolo a `.d`—: «lo malo es eso de pasar de dsa a d, NOOOO». La corrección que fijó el reparto (10/11/2026): «sc1 qué tiene que ver, es la fase 1 la que debe leer el .dsa».

13.25.1. Los tres tratamientos de la entrada, hoy

`.d` **con el lector propio** (`SCEPTRE_LECTOR=propio`): el Fortran lee tarjeta por tarjeta con `GETCK` y analiza todo, como siempre. No se tocó y es el término de comparación de todo lo demás.

`.d` **con slex** (`SCEPTRE_LECTOR=slex`, la omisión): el deck entero entra por `sceptreM/lector_slex.c` —ningún `READ` del deck queda en Fortran— y se parte en tokens con reglas declaradas. Los frentes por sección consumen los tokens directamente; lo que no tiene frente recibe tarjetas armadas de tokens y no se entera del cambio.

`.dsa` la fase 1 lo reconoce por su primer renglón (`ldsa_es_dsa`) y lo carga directo a sus variables: `libd2datdinc` (slex) arma el *dato* entero en memoria —nunca se escribe ni se lee un `.dat` ni ningún archivo intermedio— y `d_recorrer` lleva cada ítem a la rutina que guarda (`elm_valor_y_guarda`, `dp_valor_y_guarda`, `fn_equ_*`, `fn_tabla_*`, `VLUPRS`,

TABLIC, RCPRS, EXPSCN). sc1 no traduce nada: junta CASO.M.dsa + CASO.DU.dsa, separa el PARAPLOT (que es del lanzador) a CASO.graficos.ds y le pasa el reunido a la fase 1 (sceptreM/lector_dsa.c, lector_dsa_f.f90).

Por qué el parser del .dsa no emite «los tokens del .d»: eso *sería* la traducción .dsa → .d, sólo que en tokens: habría que inventar las tarjetas, sus columnas y su orden. El .dsa carga lo que el deck *significa*, no las tarjetas que lo dirían. Las variables las llenan las *hojas* de cada sección —STRELM y CMPEL (elementos), STORE sobre FCNLST (funciones y tablas), VLUPRS (el valor), CNTRL (controles), TABLIC (condiciones iniciales)— y el parser las llama en el orden en que la fase 1 las espera (el orden importa: los índices y los nombres se asignan al guardar), con entradas bind(C) como las de lector_tokens_c.f90.

En el camino del .dsa el Fortran todavía analiza texto en dos lugares: los *valores* y expresiones (CMPEL, VLUPRS, EXPSCN, carácter por carácter, como en el camino propio) y, como tarjetas servidas por un flujo de texto (lector_texto_abre de common_lector), los ítems de las secciones DC (SENSITIVITY, WORST CASE, MONTE CARLO, OPTIMIZATION: SNSPRS, WCPRS, MCPRS, OPTPRS), los parámetros G del gradiente (DPSNSP), las RERUN DESCRIPTION (sus tarjetas, que el .dsa guarda tal cual en el bloque RERUNS, a RRPRO_LEE) y los valores de más de 72 caracteres (continuaciones de GETCK). Unificar esto es lo pendiente (pasos/PENDIENTES).

13.25.2. El .d por tokens: dónde quedó cada análisis

El trabajo se hizo por etapas, cada una alternativa con opción (el lector propio nunca se tocó, los errores conservan número y firma para que el listado sea comparable) y con su vara: tests/prueba-lector.sh (los dos lectores dan el mismo código, los mismos volcados de la frontera SCEPTRE_VOLCADO, pgmsave.for, datasave.dat, acsave.dat y el listado sin fechas), tests/valores (45 decks que encuentran lo que el corpus no ve) y el corpus entero con los dos lectores y los dos modos.

Las secciones del circuito (etapa A, 09/11/2026, r7074–r7078): ELEMENTS, DEFINED PARAMETERS y FUNCTIONS analizan por tokens y llaman a la *escritura* de las hojas de siempre, partida de su análisis (elmprs_comun.f90, dpprs_comun.f90, ftnprs_comun_tabla.f90, ftnprs_comun_equ.f90); OUTPUTS e INITIAL CONDITIONS ya analizaban por tokens. Lo aprendido que hay que conservar: los analizadores de hoja miran *más allá* del final de la tarjeta (DPSNSP decide si el gradiente sigue con blnkbf(ilast+1)), así que un ítem armado de tokens deja blancos hasta el final (lector_blanquea).

Los modelos (etapa B, 09/11/2026, r7087–r7116): desde el 05/11/2026 la biblioteca no es un archivo —los modelos entran en el texto del deck y sc1 resuelve de dónde sale cada uno—, así que el deck entero, modelos incluidos, entra por slex. MDEDIT copia las tarjetas de cada modelo a una cinta de trabajo; al instanciarlo, la corriente 1 del lector (dos corrientes: la 0 el deck, la 1 el modelo) las sirve por tokens, el fin del modelo lo avisa lector_modelo_agotado() y el eco con el sufijo <model lo hace el lector, gobernado por MDPRNT. La instancia y sus opciones (modprs_slex.f90, mmodif_slex.f90) y los cambios (chgprs_slex.f90, mdelch_slex.f90, qtblch_slex.f90) van por tokens; CDPRS pregunta sólo lector_del_deck(): las tarjetas de un modelo son tarjetas como las demás. Lo aprendido: en el léxico el blanco no corta (MODEL SS1 es *un* token, el nombre viene pegado y se reconoce como prefijo); las opciones pueden seguir en el renglón de abajo; VLUPRS e IARGCK *escriben* sus argumentos (un literal revienta); el valor de un cambio termina en la coma o en el) que cierra las opciones; y la cola del lector es una sola, así que al entrar a un modelo el estado del deck se guarda entero y se repone al salir. Verificado: los nueve decks de modelos del corpus dan el mismo código, volcado y listado con los dos lectores.

La lectura 2 (etapa D, 09/11/2026, r7081–r7084): los cambios de una RERUN DESCRIPTION (ELEMENTS, INITIAL CONDITIONS, DEFINED PARAMETERS, FUNCTIONS con rerun_comun.f90, RUN CONTROLS) se despachan por tokens con las mismas reglas que sus secciones. RCTRL no se tocó: lo suyo es semántica (controles prohibidos en un rerun, omisiones de MONTE CARLO/OPTIMIZATION/WORST CASE), y su hoja RRCPRS recibe el delimitador donde lo buscaba. Lo aprendido (c08 y d04): RRCPRS no toca IXCTC cuando el token es un dato —el índice del control es el que dejó su nombre—, así que en el frente por tokens IXCTC y DATOS son inout.

Los encabezados (etapa E, 10/11/2026, r7216–r7217): cada regla de encabezado de lector_slex.c tiene su nombre en el orden de CKGRP/CKMOD, lslex_clave() devuelve el número por el *nombre de la regla* y GETCK pone NEWGRP con la clave, que viaja con el token. GETNUM y CARD no se portan: no leen el deck sino el programa generado (la cinta iqqpgm), y ahí no hay flujo de tokens que les corresponda.

El valor (etapa C, la que queda): la gramática de valor.y ya decide la forma y RUN CONTROLS ya entrega la palabra a la pieza (rcprs_bison_pal), pero el lexer de valor.y y las piezas de vluprs_bison siguen leyendo caracteres de blnkbf. Lo medido que explica el límite: los clasificadores ISEQCK, KSEQCK, IARGCK e ICSCAN *no* se pueden reemplazar por el tipo del token, porque la misma letra significa una cosa u otra según dónde aparezca (en FUNCTIONS, TABLE 2 es una tabla sólo al comienzo del renglón; en un valor, T1, Q5 o D… serían nombres), y eso lo sabe la gramática, no el léxico; además ISEQCK escribe en el buffer que lee (su chequeo de DERIVATIVE deja una marca), así que moverle el buffer es el error que se cometió en r7197 y se deshizo en r7198. Mirar los caracteres de *un token* es semántica del nombre y la necesita igual el lector propio. La herramienta que queda: SCEPTRE_V_DLIMIT=comparar calcula el delimitador de las dos maneras y avisa cuando no coinciden; encontró tres defectos en un renglón cada uno. La medida final, con tools/cuenta-texto (gdb, sin recompilar) sobre corpus/a01 —llamadas a cada analizador de caracteres—:

```
propio  dlimit 111  iseqck 49  kseqck 26  iargck 24  icscan 36  numprs 10  bcdfp 36  b2c
slex    dlimit   0  iseqck 41  kseqck 26  iargck 34  icscan  1  numprs  0  bcdfp  0  b2c
```

DLIMIT en 0 es lo buscado: con slex el Fortran ya no busca dónde termina una palabra ni dónde empieza la siguiente. Lo que queda es semántica de un token. La fase 2 no lee texto del deck (lee datasave/pgmsave, que son suyos; los DU de un .dsa los lee libdsa, en C).

13.25.3. La política de la entrada

Lo que recibe sc1 es un .d de SCEPTRE, el par partido (CASO.M.d + CASO.DU.d) o el par .dsa. **El .ds fue dado de baja como entrada** (04/11/2026): corta con el 107 y el mensaje dice cómo convertirlo *una* vez; el motivo es que traducir en cada corrida tapaba de qué archivo sale lo que corre. El rechazo de un .dsa también corta con el 107 y su diagnóstico (a diferencia de la política vieja del .ds, que seguía con el archivo tal cual y lo tapaban los errores de READIN).

De la época en que el .ds entraba traducido queda la medición grande (26/10/2026, slex r5330–r5342): la ida y vuelta .d → dato → .d sobre el corpus de SCEPTRE dio 66 iguales de 85 y *ninguno* distinto; y los 234 .ds del corpus de siru, traducidos y corridos por SCEPTRE contra siru, dieron 65 iguales, 39 distintos —casi todos lo que la traducción *avisa* que deja afuera: ecuaciones de siru, llaves, convolución, otros integradores, los circuitos de salto— y 122 que SCEPTRE no corre por sus límites (sistemas sin circuito, if() en

un valor, tensiones de nodo en expresiones, tablas de archivo). De ese trabajo quedaron en `slex/d/d.c` los renombres y arreglos que SCEPTRE exige (nombres de hasta 5/6, salidas en la forma vieja, `STOP TIME` desde una condición, `CONSTANTS` como parámetros).

13.25.4. El lector de dsa: libdsa

La decisión (04/11/2026), entre tres opciones —que `sc1` tradujera y los binarios recibieran piezas; que cada binario llamara al traductor en python; o un lector `.dsa` en C—: **el lector en C**, porque el paquete instalado no debe necesitar python3 para correr. Lo pedido, textual: «sceptre debe poder leer .dsa, el sceptreM los M y el sceptreDU los DU, y los dsa que no pueda procesar salir con error avisando». De ahí la regla del formato: lo no implementado *corta nombrando el bloque o la sentencia, con su renglón*; nada se omite en silencio.

El lector vive en slex (`slex/dsa/`), no en el árbol de SCEPTRE: un solo lugar donde está dicho qué significa cada bloque, usado por d2datdinc, siru y las dos fases. Reúsa lo que ya existía: las reglas léxicas de `dsa.slex` (un solo reconocedor de tokens, el mismo archivo que usaba el traductor en python), el *dato* de `dato.c` como representación única, y las gramáticas de cada sentencia (`rama.y`, `parametros.y`, `ecuacion.y`, `salidas.y`, …): cada bloque del `.dsa` se lleva a la sentencia `.ds` equivalente y la parsea quien ya sabe; la semántica no se reimplementa. Se construye como `libdsa.a` —estática, con `json.c` y `texto.c` adentro— y **las reglas léxicas van compiladas adentro** (`reglas_dsa.inc`): un programa enlazado no necesita ningún archivo al lado; `SLEX_DSA` elige otro.

Dos decisiones medidas que conviene recordar:

- **La fase 2 no enlaza libdsa**: el primer intento la enlazó y el modo estático dejó de compilar, porque el programa generado se enlaza a mano en once lugares y cada biblioteca nueva hay que repetirla en todos. `sceptreDU/datos_dsa.c` se lo pide a `d2datdinc --datos-sceptre`: el mismo código C, por el programa.
- **Los bloques SCEPTRE no van al texto del** `.ds` (que no tiene dónde decirlos) sino al dato, como un agregado JSON (`COTAS`, `AC`, `RERUNS`, los análisis, `SALIDAS_AC`, `SALIDAS_TARJETAS`): es el *overlay* que ya usaba el traductor, hecho dato.

La vara de la construcción fue el censo par por par contra el traductor en python (`comparar-dsa-c`, mismo `.din` salvo blancos y `@origen`): arrancó el 04/11 en **5 iguales** de 284 pares y cerró ese día en **170 iguales, 11 distintos, 6 sin leer, 12 sin vara** —los 11 «distintos» eran todos la ruta del archivo de una tabla, que es del comparador (dónde queda el intermedio de cada camino), no del lector—. Lo que la vara enseñó: `VALORES_RAMA` puede venir antes del `CIRCUITO` (las ramas se guardan y se escriben al final); los `ESTADOS_IMPLICITOS` se declaran antes que `ESTADOS`; un deck con llaves no arma su sistema sin su configuración (`--llaves`); la condición inicial viaja como `{"value": "20"}`. Y con la etapa 4 (r6629–r6631) `prueba-dsa.sh` dio sus seis puntos y el par corrió sin python, primero traducido por `d2datdinc --d` y, desde el paso 4 del plan siguiente (abajo), directo a la fase 1.

13.25.5. El camino al `.dsa` directo, y sus varas

Hasta el 16/11/2026, `sc1 caso.M.dsa` *escribía un* `.d` —`d2datdinc --d` leía el `.dsa`, lo pasaba por un `.ds` temporal y escribía `caso.d`, que la fase 1 leía—: dos traducciones de formato y dos archivos intermedios, con el rastro a la vista (el origen de una tabla salía como «TABLE 1 de .d2datdinc-dsa-1400553.ds»). El orden que dio el autor para

sacarlas: (1) los .d de la forma original al lanzador (r7248); (2) el conversor .d → .dsa en slex (dsa/escribir.c, --dsa); (3) convertir el corpus a .dsa; (4) correr el corpus completo; (5) el .dsa directo a las variables.

El escritor --dsa se escribió dos veces en paralelo, en dos sesiones que no podían hablarse, y el autor decidió *juntarlos*: el banco de ida y vuelta sobre los 293 .d del corpus daba 0 iguales con uno solo, 104 con el otro, y 203 con los dos mezclados —cada uno tenía la mitad que al otro le faltaba (el FORMATO de las TABLA de un lado; los bloques del MODELO, VALORES_RAMA, LLAVES, SEMILLAS, el INTEGRA y el corte explícito ante lo que el .dsa no sabe decir, del otro)—. La mezcla quedó en r7268. El otro consumidor del .dsa, además de SCEPTRE, es el generador mtk (siru), y lo que exige del formato quedó acordado entonces: el nodo de referencia se llama GND; el valor de una rama puede ser número, expresión de T, parámetro declarado, expresión con V/I o estados de otras ramas, llamada a una función del deck, o tabla, y con varios valores separados por comas se nombran x1, x2, …; los bloques que un lector no arma lo hacen cortar *nombrándolos* en vez de generar un sistema incompleto en silencio; y el estado que sólo está en el ic de los metadatos SIRU debe emitirse también como ESTADOS, porque sin esa línea ningún lector que no interprete ese JSON sabe que es un estado.

La fase 1 leyó el .dsa directo desde slex r8898–r8900 (libd2datdinc, d_recorrer, dsa_es_dsa) y sceptre r8901 (lector_dsa.c, lector_dsa_f.f90, la rama en sceptreM.f90). La primera vara (16/11/2026, los 104 pares del corpus por los dos caminos) dio 81 iguales, y los 23 distintos con causa conocida: 12 por los RERUN sin cargar, 8 por las secciones DC y los parámetros G sin cargar, 2 por memoria sin inicializar previa (ajena al camino: DEFECTOS 72) y 1 inválido a propósito. Ese mismo día quedaron cargados los conjuntos DC, los parámetros G, los RERUN, los valores de más de 72 caracteres y las marcas AC con sus rótulos (SALIDAS_AC), y sc1 dejó de traducir. Las varas al cierre: prueba-lector-dsa (la fase 1, .d regenerado contra .dsa directo) **103 de 104 iguales** —el restante, icmsi, era la palabra sin inicializar de DEFECTOS 72, después arreglado—; prueba-sc1-dsa (sc1 sobre el deck original contra sc1 sobre el .dsa) **78 iguales, 0 distintos**, 3 iguales sin tabla en el tiempo, 3 que no corren ni con el original, y ningún .d en el camino. Con el corpus ya en .dsa (17/11/2026) check-corpus da los 98 patrones idénticos; esas dos pruebas de comparación, cumplida su función, quedaron en viejos/tests.

13.26. El lector por tokens del .d, por dentro

13.26.1. Cómo entra slex

Con SCEPTRE_LECTOR=slex (la omisión desde el 28/10/2026; sc1 --lector propio|slex lo elige) el deck va *entero* por tokens: sceptreM/lector_slex.c lo tiene en memoria y lo entrega con la API por contexto de libslex; GETCK y GTCRD, en lugar de leer un renglón con READ, arman cada tarjeta con los tokens (lector_tarjeta: sin blancos ni comentarios en blnkbf, el renglón crudo en spltbf, como la dejaba GETCK), y todo lo que no está en slex lee así tarjetas armadas de tokens sin enterarse. Los lectores por sección que sí están (runctl_slex, otpprs_slex, ftnprs_slex, icprs_slex, dpprs_slex, elmprs_slex, dvprs_slex) los despacha CDPRS en lugar de los propios; el lector recupera por tokens la tarjeta que GETCK ya había leído (lector_retoma_tarjeta) y sigue hasta el encabezado siguiente, que deja mirado sin consumir. El eco del listado (el .d numerado) lo hace common_lector al consumir cada token, en el mismo orden y con el mismo iline; mirar (lector_mira) no ecoa.

13.26.2. Las reglas

Las tarjetas de grupo (ELEMENTS, RUN CONTROLS, OUTPUTS, FUNCTIONS, DEFINED PARAMETERS, INITIAL CONDITIONS, SENSITIVITY, OPTIMIZATION, WORST CASE, MONTE CARLO) y de modo (MODEL DESCRIPTION, CIRCUIT DESCRIPTION, CONTINUE, RE-OUTPUT, RERUN DESCRIPTION, END, RELOAD) se reconocen como las reconoce GETCK: al comienzo del renglón, sin importar mayúsculas ni blancos, las de grupo exactas y las de modo por sus primeros 12 caracteres (token de código 70, que desde la etapa E lleva también su *clave*: lslex_clave() da el número de CKGRP/CKMOD por el nombre de la regla, sin volver a mirar el texto). Una *palabra* es todo hasta un delimitador de DLIMIT —, = ( ) *— o un ; o el fin del renglón; el - queda adentro (NEWTON-RAPHSON, 1.E-3). Hay tres *estados*: el común (1); el de FUNCTIONS (2), que reconoce las cabeceras de las funciones al comienzo del renglón como ISEQCK —TABLE o T (23), DIODE TABLE (24), EQUATION / FUNCTION / Q / F (25), DIODE EQUATION (26), DERIVATIVE o D (27), DIODE solo (22); la T seguida de IME y la F seguida de REQ no son cabeceras—; y el de los conjuntos (3, grupos 8 a 11), donde la / es un token y corta la palabra.

13.26.3. Los lectores por sección, con sus errores

Cada uno conserva los números y las firmas del lector propio, para que el listado sea comparable: RUN CONTROLS (un ítem por vuelta; errores 31, 32, 22, 33, 30, 34, 86, 87 firmados RUNCTL, el 34 RCPRS); OUTPUTS (la tarjeta como unidad: la lista, los rótulos, PLOT y los modificadores AC; 22, 25, 26 de OTPPRS); FUNCTIONS (la cabecera la da slex y el cuerpo va a PRSEQU/PRSTBL; 41, 42, 43, 32); INITIAL CONDITIONS (1 a 7 de ICPRS); DEFINED PARAMETERS y ELEMENTS (slex arma cada ítem, aunque cruce renglones, y DPPRS/ELMPRS escriben); SENSITIVITY, OPTIMIZATION, WORST CASE y MONTE CARLO (dvprs_slex: cada conjunto (DEP …/ INDEP …) puede cruzar renglones; 84, 85 y 3 de DVPRS, firmados por quien llama; con la tabla llena el conjunto se lee sin guardarse, donde el propio escribía fuera de SETS).

Lo que cambia *a propósito* (lo raro se avisa, no se copia): un renglón de más de 72 columnas se lee entero por tokens (el propio ignoraba el resto, con su aviso, que el eco conserva); un control y su valor, o un elemento y sus nodos, pueden cruzar renglones; un conjunto puede cortarse antes de la / o de una coma. En el corpus ningún deck cae en esto.

13.26.4. El valor por gramática (valor.y)

Con slex, el *valor* de un elemento, de un parámetro definido, de una condición inicial o de un cambio de modelo no lo analiza VLUPRS con sus GO TO sino la gramática sceptreM/valor.y (bison); con propio, VLUPRS es el de siempre. La gramática dice la *forma* —la base (constante con tolerancia, parámetro, número, expresión, ecuación, la de un diodo, EXPRESSION o X, tabla) y lo que sigue a un * (la fuente que multiplica, o más tablas)— y lo que se escribe y los errores siguen en piezas chicas de Fortran (vluprs_bison.f90, bind(C)), que son los tramos de VLUPRS, TBLPRS, EQUPRS, EXPPRS, ARGPRS, NUMPRS y BCDFP tal cual, con los mismos mensajes y firmas. El lexer trabaja por modos según lo que la gramática espera; en la expresión da un carácter por token y sigue los paréntesis aunque crucen tarjetas. RUN CONTROLS entra por la otra puerta de la misma gramática (rcprs_bison), con la palabra ya cortada. EXPSCN quedó partido en empezar / leer (EXPPRS) / analizar, y el análisis (EXPSCN_ANALIZA) no lee el deck.

Lo raro del propio, *avisado y no copiado* (el listado cambia, a propósito; los decks están en tests/valores): NUMPRS dejaba pasar 5E, 5.5.5, 5X y 1-2; BCDFP leía 1-2, ()

como 0 % y `(9,)` con el alto en 0; el `READ` de un control leía `NAN`. Ahora todos son el error 46. Lo raro *copiado*, para que el listado sea el mismo: el segundo mensaje de `ARGPRS` con error 0 tras una hoja que falla. Lo raro que salió igual con los dos lectores quedó en `tests/valores/tablas-raras.d` (75, 74, 46 de `COMPCK`).

13.26.5. Las varas

`tests/prueba-lector.sh` (cada deck con los dos lectores: código, los dos volcados de la frontera, `pgmsave.for`, `datasave.dat`, `acsave.dat` y el listado) sobre el corpus y sobre `tests/valores` (45 decks: los que andan y uno por cada error); `check-corpus` con cada lector en los dos modos, siempre 105/105 al cierre de cada paso (r5305 a r5414, 26–27/10/2026). El "PENDIENTE" que quedaba de aquel registro —un valor de `datasave.dat` cerca de `6.95E-310` que cambiaba entre corridas— era la palabra sin inicializar de `RRBUFF`, el defecto 72, arreglado el 17/11/2026.

13.27. Los modelos expandidos en C, y las tablas en tabular2

El acuerdo (27/10/2026): la expansión de los modelos (`MODEL DESCRIPTION` e instancias) la puede hacer el C —el mismo lector de d2datdinc que la hace para siru— y SCEPTRE recibe un deck sin modelos; con `sc1 --modelos sceptre` (`SCEPTRE_MODELOS`) SCEPTRE hace todo como siempre, así se comparan las dos expansiones con el mismo lector. La regla del sufijo es *la de siru*, que es la que tiene sentido: sólo lo que el modelo define lleva el sufijo de la instancia; lo del deck y `GND`, no (SCEPTRE se lo ponía a todo, hasta a un `0` interno). Donde SCEPTRE daba otra cosa, las tablas difieren a propósito.

El orden que pidió el autor: primero la prueba, después el código. Se midió con `d2datdinc --d` que *retraducir* el deck entero no sirve como expansión (pierde rótulos, `PLOT`, modificadores AC, `RERUN`; los valores salen reescritos): lo que sirve es copiar el deck tal cual y cambiar sólo los modelos por su texto expandido. Eso fue `--d-expandido` (slex r5460–r5464), verificado con los 12 decks con modelos del corpus: 12/12 idénticos al patrón, y los 95 sin modelos byte a byte. Y después, por pedido textual («no es pasada previa; pasa lo que entra, si llega algo sin definirse se queja»), la expansión se movió *al lector mismo* (r5466): al llegar la instancia, el modelo tiene que estar leído y su texto expandido entra como un include más del flujo; un modelo no leído es error, uno recursivo se corta a las 10 000 instancias. Dos arreglos de paso que valieron también para siru: el `CHANGE` se parte en las comas de afuera de los paréntesis, y un nombre con blancos en el `CHANGE` (`TABLE 2 = TABLE 9`) ya no pierde la instancia en silencio. Con `sc1 --modelos c` (28/10/2026) los 12 decks dan lo mismo que con la fase 1.

Las tablas en tabular2 (29/10/2026): `XTABLE` (y con ella `DTABLE`) evalúa cada tabla del deck con tabular2 —la registra la primera vez y después interpola por número de vista—; es la omisión, y `sc1 --tablas sceptre` vuelve al Fortran. Decisión del autor: en una x repetida (un salto) vale el promedio de tabular2, y los patrones no se rehacen. La interpolación coincide hasta $6 \cdot 10^{-14}$, pero el integrador lo agranda: en el corpus difieren entonces una decena de casos en la sexta cifra (g02 hasta $2 \cdot 10^{-3}$).

Lo que quedó pendiente de aquel plan —qué hace `MDEDIT` con los archivos (`PERM`/`DELETE` sobre `modellib.dat`), en C o fuera de la fase 1— sigue sin decidirse: el `(PERM)` de `MDEDIT` está hoy suspendido (`pasos/PENDIENTES`).

13.28. Lo que el modelo declara para los usos: los tres programas

SCEPTRE generaba «lo que esta corrida necesita»: los conjuntos del adjunto metían fuentes en el circuito y tablas con límites literales en el generado; `RUN AC` decidía si se generaba la parte AC. Cambiar cualquier uso cambiaba el programa y había que recompilar. La decisión (28/10/2026, también en la especificación del `.dsa`): el MODELO declara `OBSERVABLES` (respecto de qué se deriva), `VARIABLES` (qué se mueve; si falta, todas) y `LINEALIZADO` (que el modelo tiene también su versión lineal), y con eso la fase 1 genera hasta *tres programas completos*: el **común** (sin adjunto ni AC), el **observable** (el común más las fuentes adjuntas de lo declarado) y el **lineal** (el común más la parte AC). `sc1` los compila aparte (`CASO-comun.so`, `-observable.so`, `-lineal.so`) y la fase 2 carga el que el uso necesita; los usos (conjuntos, objetivo, cotas, muestras, frecuencias, salidas) van en `datasave.dat` y cambiarlos no recompila. `SCEPTRE_MODELO=comun|observable|lineal` pide uno a mano; sin la variable, los `RUN` del deck siguen decidiendo, que es lo que corre el corpus.

En el deck son tres grupos del `CIRCUIT DESCRIPTION` (con el lector de slex; errores 111–113): renglones de nombres separados por comas, y `LINEALIZADO` sin renglones. Las tablas del adjunto (`KSETS`, `KDEP`, `KIND`, `KASRC`…) se leen con cuentas del `datasave` y no con límites literales, y los grupos que no hay se saltean en ejecución. El `.so` se reutiliza por la huella del *programa generado* (no del `.d`): la fase 1 corre siempre (segundos) y gfortran sólo si el programa cambió.

Con lo mismo entró **el catálogo de salidas**: `LOUTR` lleva todos los candidatos del modelo con guardas —`TIME`; de R, C y L la tensión, la corriente y el valor; de E la corriente y el valor; de J la tensión y el valor; de M el valor; los parámetros definidos, los W del AC, las X escalares internas y las derivadas— y la fase 2 resuelve `OUTPUTS` por nombre: cambiar las salidas no cambia el programa.

Las pruebas viven en `corpus/separar/` (los pares declarados de a08, a10, d04 y a11, los decks de error del lector) con `viejos/tests/prueba-separar.sh` y `prueba-separar-sc1.sh`; la aceptación del 28/10: los modelos completos dan el mismo listado que sus decks base, con sensibilidades de a08 a menos de $1{,}3 \cdot 10^{-6}$ relativo (aceptadas por el autor), y el corpus entero sin cambios.

13.29. La reentrancia: varios circuitos en un mismo proceso

13.29.1. La instancia, por intercambio

Sacado `-fno-automatic`, quedaba *una* copia del estado: un proceso, un circuito. La primera idea —cada variable de módulo como `POINTER` al componente del circuito activo— **no sirvió**: SCEPTRE pasa en todos lados un elemento como comienzo de arreglo (asociación de secuencia), que el estándar no permite con un arreglo `POINTER`. Lo que quedó es *el intercambio* (`herramientas/intercambio.py`): las variables de cada módulo siguen como estaban y son siempre las del circuito activo; cada módulo gana un tipo `inst_<m>_t` con las mismas variables, un arreglo de instancias y `guarda_`/`trae_` (con `move_alloc` para los allocatables: no copia), y `circuito_activa(k)` cambia todo junto. Los módulos con `EQUIVALENCE` guardan el arreglo de fondo una vez, no cada nombre. Qué es de la instancia y qué del proceso: los valores, tablas, controles, integradores y el adjunto van a la instancia; las unidades de cinta, la tabla del `.so` cargado y el latido eran del proceso (después también pasaron al circuito, abajo).

13.29.2. La fase 1 reentrante

El contrato entre leer y generar se midió primero (`herramientas/frontera-fase1.py`): 169 variables de 30 módulos que la lectura escribe y la generación lee, volcadas en orden fijo con `SCEPTRE_VOLCADO` justo antes de `GENER8` —la prueba de la reentrancia y la vara de los dos lectores—, y un segundo volcado (`.2`) al terminar la lectura 2. Las etapas, con lo que cada vara enseñó: `sceptreM --varios` (un deck por directorio en el mismo proceso) dio al principio **0 pares iguales de 80** —el segundo deck heredaba la tabla de elementos del primero—; con la copia por circuito de 47 módulos más lo que cada rutina recuerda (`common_rec1_*`, `herramientas/recuerda1.py`) pasó a 78/80, y con el defecto 57 arreglado (`TOTAL_NOMBRES` sin inicializar) a **80/80**. La *salida única*: `SC1_SALIR` anota y *vuelve* (1326 vueltas puestas por `herramientas/salida-unica.py` tras cada llamada que puede salir), con lo que un error ya no termina el proceso; y `sceptreM --lote` corre cada deck en un hijo con su nombre adelante de cada renglón. La *partición de la lectura 2* (RERUN/CONTINUE/RE-OUTPUT) separó `RRPRO_LEE` de `RRPRO_GENERA` (los grupos completos a `common_corridas`, sin límite), `WTDAT_PREPARA` entre las dos lecturas, y `CNTNU/XOTPR` en `_LEE` y `_GENERA`: el orden que cualquier lector debe respetar es leer el deck (volcado 1) – `GENER8` – `WTDAT_PREPARA` – leer la lectura 2 (volcado 2) – generar el resto. Verificado byte a byte contra un `sceptreM` de antes sobre los ocho decks con RERUN.

13.29.3. La fase 2 reentrante, hasta alternar paso a paso

El mismo camino sobre x2: `EXEC2_CORRE` como rutina y `sceptreDU --varios` midió 51 pares iguales de 82; cada circuito desde cero (los `olvida_`, las unidades abiertas y cerradas por circuito, el `fort.7` que `POS4` dejaba abierto) lo llevó a 81; la salida única de `falla_corta` (y el generado, al que `XF90` le pone `IF (SC2_SALIENDO()) RETURN` tras cada llamada) a **82/82**; y el mismo `.so` para dos circuitos se protege cargando una copia con otro nombre. La etapa grande —varios circuitos *vivos*, alternados paso a paso— partió el lazo del generado: `SIMUL8` de a un paso (`SC2_PAUSA`), un juego de punteros al generado por circuito, las unidades con `newunit`, y `sceptreDU --alternar DIR…`; `prueba-varios2` cerró en **170/170** (82 pares seguidos, 82 alternados, 4 consigo mismos) con el corpus intacto.

13.30. El circuito como objeto dinámico, y el servidor

Las decisiones (19/10/2026): la fase 1 produce `CASO.so`; la fase 2 es *un* ejecutable fijo que lo carga con `dlopen` y no se enlaza nada por circuito; el par `CASO.so` + `CASO.datasave` se guarda junto al deck y se reutiliza por huella; los módulos emitidos viven dentro de cada `.so`. Lo medido antes de planificar: el deck lo lee sólo la fase 1; x2 llama *por nombre* a nueve unidades del generado (`SIMUL8`, `SIMIC`, `SIMTR`, `LOUTR`, `SIMAC`, `CDCALC`, `KKCALC`, `AGDIM`, `LSDIM`) más `SC1_TAMANIOS`; y `datasave.dat` no es inmutable: la fase 2 lo reescribe al terminar (lo que hereda un `CONTINUE`).

Las seis tandas de un día (r4149–r4179): los *despachadores* (`common_generado_2`: un puntero a procedimiento por entrada, `dlsym` y `c_f_procpointer`; en modo estático el generado sigue pisando las cáscaras al enlazar, que fue la verificación intermedia); `sceptreDU` ejecutable (con `SCEPTRE_SO` carga; códigos 83 no se pudo cargar, 84 falta un símbolo, 85 la huella no coincide); `sc1` compila a `.so` junto al deck y reutiliza por huella (`-rehacer` lo fuerza; `-Bsymbolic-functions`, porque `-Bsymbolic` a secas colgaba a08); `SC1_ORIGEN` (las huellas del deck, de la modellib y de x2, comparadas por las dos fases); el contrato en `docs/INVOCACION-SCEPTRE.md`; y la prueba (`tests/prueba-so.sh`). La tanda 7 (pedido

del autor) pasó libsceptreDU a libtool: `libsceptreDU.la`, `sceptreDU` enlazado contra la `.so` y el `.so` del circuito resolviendo contra ella.

El servidor (`sceptreDU --servidor`): lee trabajos de la entrada estándar —`DIRECTORIO CASO.so [VAR=VALOR…]`— y contesta `DIRECTORIO CODIGO` por renglón (86 el directorio no existe, 87 el renglón no se entiende; 128 más la señal si lo mató una señal). La alternativa elegida: el padre carga el `.so` (queda cargado para el próximo) y hace un hijo por trabajo, así cualquier falla termina el hijo y el servidor sigue; `--sin-hijos` corre en el mismo proceso con `EXEC2_CORRE` (sin la protección de las señales). Desde `sc1`, `-lote` lo usa por dos FIFO (capítulo del flujo). La prueba (`tests/prueba-servidor.sh`): optimización, convolución, Monte Carlo y el deck que se cuelga (118), con un directorio inexistente en el medio, todos con el mismo resultado que corridos solos.

13.31. La memoria dinámica: cómo desaparecieron los topes

La condición que mandó sobre el diseño (el autor, 18/10/2026): «el futuro objetivo es hacer que la fase uno produzca un módulo dinámico que el dos use sin linkear»; los tamaños no pueden fijarse al compilar la fase 2. **El diseño en una frase: el generado le dice a la fase 2 cuánto mide todo, antes de que la fase 2 reserve nada.** Lo medido primero: la fase 2 ya era «dinámica» en la forma y no en el fondo (66 allocatables, todos alocados al máximo fijo de 300, porque el orden estaba al revés: dimensionaba antes de que el generado cargara los valores); los 51 punteros del módulo emitido se ligaban *en la declaración*, lo que no puede funcionar contra un `.so`; y la fase 1 no tenía nada dinámico, con `EQUIVALENCE` por todos lados y la dificultad de que lee el deck renglón a renglón sin saber cuántos elementos vendrán.

Las tandas: `SC1_TAMANIOS` (una *subrutina* emitida en el generado, a propósito —un ejecutable puede hacer `dlsym` de una subrutina, no leer un parámetro de un módulo que no conocía—); `sceptreDU` pregunta primero y reserva después (un tamaño por clase de arreglo, no por circuito; `-fcheck=bounds` como vara, que también destapó el «idioma de 1966»: dummies de largo fijo 2 recorridos hasta N, pasados a `(*)`); las ligaduras como sentencias ejecutables que `READIN` ejecuta al entrar; el disperso y el GEAR; y la fase 1 entera en once sub-tandas (5a–5l): `CONTAR` (la pasada de conteo, decisión del autor: *las dos cosas*, la cota del conteo como primera reserva y crecer con `move_alloc` si aun así se llena, con lo que los «table overflow» quedaron inalcanzables), los `EQUIVALENCE` deshechos con vistas por `c_f_pointer`, las listas de `STORE` que se agrandan y reintentan, el buffer de ecuaciones, las tablas dispersas a `INTEGER*4`. La llave `SCEPTRE_COTAS_MINIMAS` (todas las cotas en 1) hace que el corpus entero ejercite el crecimiento, y tiene que dar lo mismo —y da—.

Quedaron en el corpus los decks que prueban que el tope no está: `rcgrande` (321 elementos, contra la escalera RC calculada aparte), `dpgrande` (250 parámetros), `outgrande` (120 salidas, que pidió el tercer formato del `.dirac`, autodescriptivo, decidido «un solo formato nuevo, sin límite adentro»), `dispgrande`. La medición final que vale recordar: la ganancia no es memoria (el residente hasta sube, porque lo dinámico se toca y lo fijo en `.bss` no se contaba), *es que ya no hay tope*. Dos hallazgos de paso: el 105 (falta un archivo) le ganaba al motivo real del rechazo, corregido ese mismo día; y `dispdensa` da mal desde 68 capacitores *también en el original* (el buffer fijo de la ecuación del nodo común, ya dinámico; el resto es un defecto heredado, no de la memoria).

Lo que sigue fijo, anotado en `pasos/PENDIENTES`: más de 999 elementos en la formulación vectorial (los nombres `X(nnn)` son de 6 caracteres; decisión del autor: pendiente, la fase 1 sale con el 91 en vez de generar un programa que no compila), y los fijos chicos que quedaron (`common_tbles` 200, `jqqorq` 100, la tabla dispersa de la fase 1).

13.32. La modernización del Fortran, y por qué hacía falta

13.32.1. Las construcciones borradas

Con `-std=gnu` —el modo normal del gfortran moderno— el árbol heredado compilaba, pero avisaba 858 veces, y *todos* los avisos eran «Fortran 2018 deleted feature»: lazos `DO` que cierran en una sentencia ejecutable etiquetada, `IF` aritmético y `DO` compartido. Se eliminaron los 858 (el inventario, una línea por aviso, quedó en `pasos/borradas-x1.inv` y `borradas-x2.inv`) y el árbol dejó de necesitar `-std=legacy`.

13.32.2. El susto de `-O2`, que era un síntoma

En octubre de 2026 el empaquetado —que compila con los flags de Debian, `-O2`— dio *números distintos* de los patrones. No era redondeo: con `a01` el integrador daba 1849 pasos en lugar de 3036 y la peor diferencia relativa llegaba al 142 % (582 % en `a06b`). Sobre el corpus: 33 idénticos con `-O0`, nueve con `-O2`. Medido directorio por directorio, la fase 1 toleraba la optimización y la fase 2 no.

La causa era el idioma de los COMMON: un arreglo declarado `COMMON /NAMES/ V(1)` y recorrido mucho más allá de 1, porque en realidad es una *ventana* sobre un bloque que otra rutina declaró entero. Eso es comportamiento indefinido, y el optimizador de bucles lo aprovecha al revés: si un lazo accede `V(i)` con $i > 1$ en un `V(1)`, concluye que no puede iterar más de una vez y lo recorta. La prueba: `-fno-aggressive-loop-optimizations` —el flag que prohíbe justamente esa suposición— devolvía 25 de los 33. No se eligió ese flag como solución: «un paquete que da bien casi todo es peor que uno lento que da bien todo», así que se compiló sin optimizar *y se atacó la causa*. Quedaron ocho casos sin explicar con el flag puesto, y se descartó la contracción FMA, las trece pasadas que `-O1` agrega sobre `-Og` (una por una) y las variables sin inicializar.

Hoy la historia está cerrada: sacado el idioma —arreglos con su dimensión de verdad, y después allocatables (agregado 13.31)— los patrones salen idénticos con `-O2`, que es el valor por omisión de autoconf y el que usa el árbol. El único aviso que `-O2` agrega es un `-Wmaybe-uninitialized` del evaluador de expresiones de `sd2`, que es código de terceros y no se silencia: taparlo esconderia los de verdad.

13.32.3. Los COMMON, bloque por bloque

Primero las 140 declaraciones `V(1)` pasaron a llevar su máximo —de los límites que la fase 1 ya tenía, no inventados— y los 77 argumentos `(1)` a `(*)`, que es la forma legal de decir «el tamaño lo sabe el llamador»; eso solo arregló cuatro casos a `-O2`. Después los bloques se fueron, de a uno, a módulos.

La lección que costó un intento deshecho: se probó mover cinco bloques de la optimización a un módulo común de una vez; compiló sin avisos y el corpus se rompió, porque varias rutinas contaban con que un COMMON estuviera *al lado* del otro en memoria —la asociación de secuencia entre bloques, que en un módulo ya no vale—. Desde entonces la regla fue **de a un bloque por tanda**, con el corpus idéntico antes de seguir. Así se eliminaron los 65 bloques; doce de ellos resultó que el programa generado los declaraba y no los tocaba nunca, y la fase 1 dejó de emitir esas tarjetas.

13.32.4. Los `ENTRY` y los `GOTO`

La reentrancia mostró dónde estaba el estado escondido: de los cuatro errores que dio sacar `-fno-automatic` en la fase 1, dos eran rutinas con `ENTRY` que usaban en una entrada

lo que había dejado la otra —un `ENTRY` es un cuerpo compartido: las locales son de todos y lo que una entrada deja, otra lo lee sin que se vea—. Había 13 `ENTRY` en 8 fuentes y unos 4000 `GOTO`; ninguno cruzaba rutinas (no hay retorno alternativo en ninguna fase). Los `ENTRY` se fueron todos; los `GOTO` se desarmaron donde escondían el flujo, que es lo que la reentrancia necesitaba poder ver.

13.32.5. El Fortran que el SCEPTRE *escribe*

Un intento del 12/10/2026 quedó **revertido** y se documenta para quien lo retome: el programa que la fase 1 *genera* (`pgmsave.for`) todavía usa construcciones borradas —lazos `DO` que cierran en una sentencia etiquetada, y el `IF` aritmético que `CVPREP` emite—, y por eso `sc1` lo compila aparte con `-std=legacy`. Modernizar al *generador* es trabajo de quien retome el generado; la marca del emisor viaja en la columna 73 de cada renglón generado, que es por donde se mide quién emite qué.

13.33. Las clases del SCEPTRE, y qué prueba el corpus

13.33.1. Qué son las clases

SCEPTRE no resuelve el circuito sobre la lista de elementos: primero arma un *árbol* del grafo y reparte los elementos en **clases** según el tipo y según si quedaron en el árbol o fuera. Las ecuaciones de estado salen de ese reparto, y casi toda la fase 1 está organizada por clase: `NCLENT(8)` dice cuántos cayeron en cada una, `ISTART(8)` dónde empieza cada una en la tabla, y las submatrices `B14`, `B25` y `B36` dicen qué clases quedaron *acopladas*. Las guardas de la fase 1 miran eso, no el deck: `ICTEQS`, por ejemplo, emite las matrices de condiciones iniciales de la clase 3 sólo si `B36` no es cero.

La confusión que costó tres pruebas, y conviene leerla antes de escribir un deck para ejercitar un camino: la tabla que el listado imprime bajo `INDICES FOR STATE VARIABLE DERIVATIVE NAMES` no es el reparto por clases del árbol. Un deck que «tiene clase 3» en ese listado puede no estar tocando el camino que se quería tocar; lo que decide es el árbol, y para forzar un acoplamiento hay que construir la topología, no elegir los tipos de elemento.

13.33.2. Qué parte del código no prueba el corpus

Medido con gcov sobre todos los casos con patrón, las dos fases, con las cadenas de las continuaciones: de 335 fuentes, **once** no los toca ningún deck, y los once son *imposibles* de cubrir con cualquier deck —no son huecos del corpus—: dos son módulos de sólo declaraciones (su 0 % es un artefacto de la medición) y cinco son las *plantillas* de las secciones que se generan (`LOUTR`, `SIMAC`, `SIMIC`, `SIMTR`, `PDRV1`: el código que corre es el del `pgmsave`, no estas copias).

Para los caminos que sí se podían cubrir y ningún ejemplo recorría, se estudió en el código qué condición de la *entrada* activa cada rutina —quién la llama, hacia arriba, hasta la tarjeta o la sección del deck— y se escribieron decks a propósito, verificados con gcov y versionados en el corpus con su patrón: `listados` (`LIST NODE MAP` y `WRITE DEBUG`), `acdep` (`RUN AC` con fuente dependiente lineal), `jacimplic` (lazo de C, corte de L, R variable por tabla, parámetros con derivada parcial e `INTEGRATION ROUTINE = IMPLICIT`) y los demás que el corpus fue sumando. Esa es la manera de agregar cobertura en este árbol: no escribir un deck «grande», sino uno que active por topología la guarda que se quiere ejercitar.

13.34. El plan de modernización, paso por paso

El trabajo de este árbol se organizó en doce pasos sobre la copia del último SCEPTRE de Novender. **Están todos hechos**; la tabla dice dónde está contado cada uno, porque es el mapa para entrar al árbol.

0. Compilar con los Fortran modernos. Las 858 construcciones que Fortran 2018 borró, eliminadas: ya no hace falta `-std=legacy` (agregado 13.32).

1. La salida `.sim` en texto. La fase 2 escribe el `.sim` ASCII además del `.dirac` binario, con el formato que Simusol consume (capítulo 3).

2. Reemplazar `FCreadline`. Resuelto de fábrica en el 26.131: `sd2` ya usa GNU readline con historial; el configure lo detecta (`READLINE_LIBS`).

3. Las opciones de línea de órdenes. Unificadas en `sc1`, con su página de manual (`docs/sc1.1`) y la configuración por usuario (capítulos 2 y 8).

4. Makefiles tipo GNU. Autoconf, automake y libtool, con el *check real* del corpus (capítulo 11).

5. Memoria dinámica y BLAS/LAPACK. Los topes fijos y los COMMON, afuera (agregados 13.31 y 13.32); el álgebra por LAPACK/BLAS y UMFPACK por omisión (capítulo 7).

6. Lo positivo de los árboles anteriores. Lanzadores reubicables, `sceptre-user-setup`, el empaquetado desde el `.summa` y los mapas regenerables. Sin `ngp`/gnuplot, que el autor descartó. El empaquetado *no* compila en `postinst`: compila al construir, y gfortran queda como dependencia de ejecución porque `sc1` compila en cada corrida.

7. Documentar. Las cuatro páginas de manual (`sc1`, `sceptre`, `sceptre-user-setup`, `sd2`), este libro y la traducción del manual histórico.

8. slex/bison para leer los `.d`. El lector por tokens, el valor por gramática y los modelos expandidos en C (agregados 13.26 y 13.27).

9. Los modelos como módulos dinámicos. El circuito como `.so` y el `sceptreDU` que queda corriendo (agregado 13.30).

10. Sacar los `ENTRY` y los `GOTO`. Hecho; fue lo que destapó el estado escondido de la reentrancia (agregados 13.32 y 13.29).

11. Separar modelo, datos y usos. La política y lo que corre, en el capítulo del lenguaje; el `.dsa` leído directo por la fase 1, en el agregado 13.25.

13.34.1. Dos cosas que el plan anotó como límites y dejaron de serlo

Conviene recordarlas porque las dos parecían de fondo y no lo eran:

- **«Una fase 2 trabada queda colgada»**, y la AC de cientos de ramas que se colgaba en el modo estático y terminaba en el dinámico *con el mismo generado*: se anotó como razón para preferir el modo dinámico, y era un defecto —`IWORK` y `LOCAL` de `EIGEN1` eran de 100 fijos y `REALV1` escribía más allá; en estático pisaba su propio contador y en dinámico otra cosa, dando un 81 o un 125 *falsos*—. Arreglado, los dos modos dan lo mismo. Y desde entonces las dos fases *laten*: si no avanzan, guardan lo hecho y salen con el 118 diciendo dónde estaban.

- **«El QR no converge con 1200 estados, hace falta LAPACK»**: no era el QR —la matriz es diagonal y `HESQR1` ni itera— sino un ancho fijo de los nombres de 999, que hacía escribir fuera de dos arreglos y pisar la matriz. LAPACK quedó como mejora, que después se hizo, no como necesidad.

A Códigos de salida

Las dos reglas del autor, que mandan sobre todo: **el código dice si hubo un fallo, no si hubo simulación** (simuló, o no simuló y no debía: 0; lo demás, el código del fallo); y **lo extremo corta, lo demás se junta y se reporta todo de una** (un error del deck se anota y se sigue leyendo; un fallo numérico extremo o un error interno cortan). Los números son *uno por punto de salida*, todos distintos, cada uno en el rango de su familia; la agrupación por causa es tabla del lanzador, no del programa. `STOP` para lo que es culpa del deck; `ERROR STOP` (con backtrace) para lo interno. `--parar 1|2` elige entre seguir mientras se pueda (la omisión) y parar en el primer fallo; `--verbose` gradúa el listado sin tocar el defecto, que es byte a byte el de los patrones.

El resumen por familias:

0	terminó lo que se pidió (simular, o no había que simular)
5–9	archivos y base de resultados
10–29	una tabla se desbordó (casi vacío: la memoria dinámica crece)
30–39	el área de trabajo no alcanza
40–69	no se encontró algo que el deck nombra
70–89	estados que no se saben manejar, e internos (83–85: el .so; 86–87: el servidor; 88–89: `SCEPTRE_DATOS`)
90–99	lo no implementado y la memoria (92 CALCULA, 93 EXPRTK, 95 memoria, 96 cuota de disco)
101	el deck tiene errores, y están todos en el listado
102	la preparación no se completó
104	el circuito es contradictorio o imposible
106–109	el guion: línea de órdenes, entrada, apertura, cadena
111–128	los fallos de la simulación (118 el latido, 120/111 singular, 121/114 Newton, 122–123 condiciones iniciales, 124/115 paso)

La tabla, número por número

Registrada al cerrarse el trabajo de los códigos (14–21/10/2026) y mantenida con el criterio «primero la familia, después el número»: un número nuevo que no entre en ninguna familia es señal de que falta una familia. Fase y rutina son las del punto representativo.

código	**fase**	**rutina**	**qué pasó**
4	1	GETCK/GTCRD	no se pudo leer el deck (E/S, no el fin)
5	2	POSWR1	reservado (era el directorio del .dirac lleno; el tercer formato crece)
6	2	POSOPEN	no se pudo abrir la base de resultados
7	2	POSWREC1	no se pudo escribir su bloque de directorio
8	2	—	reservado (un close que falle)
9	2	POSRREC1	no se pudo leer su bloque de directorio
10/11	1	GROW/GNLOCA	la tabla dispersa se desbordó
12	1	STORE	se excedió la capacidad
13	1	STOEQN	la tabla de ecuaciones se desbordó
14	1	SC1MOD	no caben más declaraciones de módulo emitido
15	1	CKCONV	más de 50 núcleos de convolución

16	2	—	reservado (el bloque de listado ya mide lo que haga falta)
17	2	RDCONV	la tabla de un núcleo no entra en el buffer
18	2	ETABLE	una tabla de tabular2 falló (el mensaje es el de tabular2)
30	1	AUGM	el arreglo no entra en el área de trabajo
40–49	1	EQFORM…WTEQTN	no se encontró un término, fuente, resistor, elemento, variable (46), o la ecuación no se pudo escribir (49)
70–76	1	BVSCAN…VMUTUL	un estado que el programa no sabe manejar
77–82	2	MXMULT…CVYCLC	errores internos de la fase 2
83/84/85	2	sceptreDU	el .so no carga / le falta una entrada / su huella no coincide
86/87	2	servidor	el directorio del trabajo no existe / el renglón no se entiende
88/89	2	SC2_DATOS	un nombre que el modelo no tiene / el archivo no se pudo leer
90	1	EQFORM	expresión de valor del tipo 6: no implementada
91	1	XIBCD/CPYFIJO	un número del generado no entra en sus cifras (antes: el tope de 999 de la forma vectorial)
92	2	CALCULA	el evaluador externo falló (no arrancó, ERR, timeout)
93	2	EXPRTK	la expresión no compila, o la tabla se agotó
95	1	TREE	no se pudo reservar memoria
96	1	CUOTA	no queda lugar donde la corrida escribe (una escritura que no entra no vuelve con error: libgfortran informa éxito y pierde los datos)
101	1	SCEPTREM	el deck tiene errores, y están TODOS en el listado
102	1	SCEPTREM	no se puede simular: la preparación no se completó
104	1	SCEPTREM	el circuito es contradictorio o imposible (lazo de fuentes, topología que prohíbe las IC)
106/107/108/109	–	sc1	línea de órdenes / la entrada no sirve / no se pudo abrir o falta una pieza / falló la cadena
110	1	error()	la severidad 4: corta en el acto
111–115	2	INVERS…EXPO	singular / autovectores / XMR / el Newton agotó sus pasadas / las derivadas piden un paso menor
116/117	2	LSNEW	no se encontraron los autovalores / un autovector
118	1,2	LATIDO	no avanzó en SCEPTRE_SIN_AVANCE segundos: guarda lo hecho y dice dónde
120–128	2	LINSOL…FCONVY	los nueve fallos de la simulación del contrato con el árbol viejo (singular, Newton, paso mínimo, inestable, IMPLICIT cancelada, el adjunto, la convolución)

Un corte de la fase 2 durante la simulación guarda lo hecho igual que el latido: los puntos ya calculados, el `.dirac` y el listado. Los tres fallos que cortan *a propósito* aun con `--parar 1` —inestable, la IMPLICIT cancelada, la convolución mal llamada— cortan porque quien llamó espera un resultado que no se pudo calcular. El 126, 127 y 128 están en la zona que el shell usa para los suyos: vienen del contrato con el árbol viejo y un lanzador debe saber que, si `sceptreDU` corrió, son de SCEPTRE.

El **103** y el **105** ya no se usan: eran «faltó `modellib.dat`» y «faltaba pero la corrida siguió

con uno vacío», y la biblioteca de modelos dejó de ser un archivo que puede faltar (5 de noviembre de 2026, véase 13.24). Los números quedan libres.

Ninguno pasa de 125: de 126 para arriba el shell los confunde con los suyos.

B Ejemplos

B.1. La descarga que resuelve un motor: calculaohm y exprtkohm

Un capacitor de 1 F con tensión inicial 1 V se descarga por una resistencia de 10 Ω que *no* está escrita como elemento: la corriente la da una función del deck, resuelta por un motor. La solución exacta es $V_{C1}(t) = e^{-t/RC} = e^{-t/10}$, y por eso el par sirve de patrón de los motores. Son los casos `calculaohm` (implícito, por calcula) y `exprtkohm` (explícito, por ExprTk) del corpus.

El deck SCEPTRE de calculaohm (`viejos/corpus/calculaohm.M.d` + `.DU.d`):

```
MODEL DESCRIPTION
  MODEL DESCARGA (A-B)
    ELEMENTS
      JD, B-A = FI
    DEFINED PARAMETERS
      PRD = 10
    FUNCTIONS
      FI = CALCULA ('calculaohm.din', I, V=VJD, R=PRD)
      FP = CALCULA ('calculaohm.din', P, V=VJD, R=PRD)
CIRCUIT DESCRIPTION
CALCULAOHM : DESCARGA DE UN CAPACITOR POR UNA RESISTENCIA QUE RESUELVE CALCULA
  ELEMENTS
    C1, 1-2 = 1
    D1, 1-2 = MODEL DESCARGA
  INITIAL CONDITIONS
    VC1 = 1
  OUTPUTS
    VC1, FI
  RUN CONTROLS
    STOP TIME = 5
    INTEGRATION ROUTINE = TRAP
END
```

El módulo que calcula resuelve (`corpus/calculaohm.din`): dos salidas a propósito, para probar que las `F` que comparten módulo y entradas se junten en un solo `EVAL`.

```
variables
  v = 1 [V]         # tension impuesta
  r = 10 [ohm] > 0  # resistencia
  i = 0.1 [A]       # corriente que resulta
  p = 0.1 [W]       # potencia disipada
ecuaciones
  v = i*r           # ley de Ohm
  p = v*i
```

El mismo modelo está también en `.dsa` (`corpus/calculaohm.M.dsa`, escrito a mano: `PARAMETROS: V, R`, las dos ecuaciones residuales y sus `SEMILLAS`), y verificado por el camino de siru:

```
siru --calcula-puntos --du-stdin calculaohm.M.dsa < calculaohm.DU.dsa
   -> i = 0.1, p = 0.1
```

`exprtkohm` es la misma descarga con la ley de Ohm despejada: `FI = EXPRTK ('v/r', V=VJD, R=PRD)`, sin proceso externo.

La corrida entera por `sc1` queda pendiente de que la fase 1 lea `CALCULA` y `EXPRTK` en `FUNCTIONS` (sección 4; `pasos/PENDIENTES`): por eso estos dos casos del corpus no tienen patrón todavía, y la comprobación contra $e^{-t/10}$ se hará al cerrarse eso.

Índice alfabético

Ajolote, capacidad, 41

clases, 62
cobertura, 62
COMMON, 61

`.dsa`, 51

Fortran 2018, 61
funciones del usuario, 45

instancia, 58

lector, 51
lector por tokens, 55
LINEALIZADO, 58

memoria dinámica, 60
MODEL EDIT, 50
modellib.dat, 44, 47
modelos
    expansión en C, 57
modelos propios, 47

OBSERVABLES, 58
optimización, 61

plan de modernización, 63

reentrancia, 58

sceptre-user-setup, 42
servidor, 59
slex, 51
`.so`, 59

tabular2, 57

valor.y, 55