Tesis de la Carrera de
Doctorado en Ingeniería Nuclear

Transporte de neutrones en la nube

Mg. Ing. Germán Theler
Doctorando

Dr. Fabián J. Bonetto
Co-director

Dr. Alejandro Clausse
Co-director

San Carlos de Bariloche
Octubre 2023

Instituto Balseiro
Universidad Nacional de Cuyo
Comisión Nacional de Energía Atómica
Argentina

A mis increíbles hijos Tomás y Máximo
y a mi esposa Natalia,
que me han acompañado en esta montaña rusa.

Y a todos los que entrenan laterales con sandías.

Resumen

Transporte de neutrones en la nube

- state the problem
- say why it's an interesting problem
- say what your solution achieves
- say what follows from your solution

Palabras clave transporte de neutrones, difusión de neutrones, computación en la nube, computación de alto rendimiento, elementos finitos, mallas no estructuradas

Revisión 2023-10-07–c82918e+dirty

Abstract

Neutron transport in the cloud

- state the problem
- say why it's an interesting problem
- say what your solution achieves
- say what follows from your solution

Keywords core-level neutron transport, neutron diffusion, cloud computing, high-performance computing, finite elements, unstructured grids

Revision 2023-10-07–c82918e+dirty

Tabla de contenidos

Apéndices 59

1. Implementación computacional

> C++ is a horrible language. It's made more horrible by the fact that a lot of substandard programmers use it, to the point where it's much much easier to generate total and utter crap with it. Quite frankly, even if the choice of C were to do *nothing* but keep the C++ programmers out, that in itself would be a huge reason to use C.
> [...]
> C++ leads to really really bad design choices. You invariably start using the "nice" library features of the language like STL and Boost and other total and utter crap, that may "help" you program, but causes:
>
> - infinite amounts of pain when they don't work [...]
> - inefficient abstracted programming models where two years down the road you notice that some abstraction wasn't very efficient, but now all your code depends on all the nice object models around it, and you cannot fix it without rewriting your app.
>
> [...]
> So I'm sorry, but for something like git, where efficiency was a primary objective, the "advantages" of C++ is just a huge mistake. The fact that we also piss off people who cannot see that is just a big additional advantage.
> *Linus Torvalds*, explaing why Git is written in C, 2007

> C++ is a badly designed and ugly language.
> It would be a shame to use it in Emacs.
> *Richard M. Stallmann*, explaining why Emacs is written in C, 2010

Hay virtualmente infinitas maneras de diseñar un programa para que una computadora realice una determinada tarea. Y otras infinitas maneras de implementarlo. La herramienta computacional desarrollada en esta tesis, denominada FeenoX (ver **?@sec-faq** para una explicación del nombre), fue diseñada siguiendo un patrón frecuente en la industria de software:

1. el "cliente" define un documento denominado *Sofware Design Requirements*, y
2. el "proveedor" indica cómo cumplirá esos requerimientos en un *Sofware Design Specifications*. Una vez que ambas partes están de acuerdo, se comienza con el proyecto de ingeniería en sí con *kick-off meetings*, certificaciones de avance, órdenes de cambio, etc.

El SRS para FeenoX (Apéndice A) es ficticio pero plausible. Podríamos pensarlo como un llamado a licitación por parte de una empresa, entidad pública o laboratorio nacional, para que un contratista desarrolle una herramienta computacional que permita resolver problemas matemáticos con interés práctico en aplicaciones de ingeniería. En forma muy resumida, requiere

a. buenas prácticas generales de desarrollo de software tales como

   - trazabilidad del código fuente
   - integración continua
   - posibilidad de reportar errores
   - portabilidad razonable
   - disponibilidad de dependencias adecuadas
   - documentación apropiada

b. que la herramienta pueda ser ejecutada en la nube

   - a través de ejecución remota
   - que provea mecanismos de reporte de estado
   - con posibilidad de paralelización en diferentes nodos computacionales

c. que sea aplicable a problemas de ingeniería proveyendo

   - eficiencia computacional razonable
   - flexibilidad en la definición de problemas
   - verificación y validación
   - extensibilidad

Observación. El requerimiento de paralelización está relacionado con el tamaño de los problemas a resolver y no (tanto) con la performance. Si bien está claro que, de tener la posibilidad, resolver ecuaciones en forma paralela es más eficiente en términos del tiempo de pared[1] necesario para obtener un resultado, en el SRS la posibilidad de paralelizar el código se refiere principalmente a la capacidad de poder resolver problemas de tamaño arbitrario que no podrían ser resueltos en serie, principalmente por una limitación de la cantidad de memoria RAM.

Observación. Si bien es cierto que en teoría un algoritmo implementado en un lenguaje Turing-completo podría resolver un sistema de ecuaciones algebraicas de tamaño arbitrario independientemente de la memoria RAM disponible (por ejemplo usando almacenamiento intermedio en dispositivos magnéticos o de estado sólido), prácticamente no es posible obtener un resultado útil en un tiempo razonable si no se dispone de suficiente memoria RAM para evitar tener que descargar el contenido de esta memoria de alta velocidad de acceso a medios alternativos (out-of-core memory como los mencionados en el paréntesis anterior) cuya velocidad de acceso es varios órdenes de magnitud más lenta.

[1] Del inglés *wall time*.

Observación. Este esquema de definir primero los requerimientos y luego indicar cómo se los satisface evita un sesgo común dentro de las empresas de software que implica hacer algo "fácil para el desarrollador" a costa de que el usuario tengo que hacer algo "más difícil". Por ejemplo en la mayoría de los programas de elementos finitos para elasticidad lineal es necesario hacer un mapeo entre los grupos de elementos volumétricos y los materiales disponibles, tarea que tiene sentido siempre que haya más de un grupo de elementos volumétricos o más de un material disponible. Pero en los casos donde hay un único grupo de elementos volumétricos (usualmente porque se parte de un CAD con un único volumen) y un único juego de propiedades materiales (digamos un único valor E de módulo de Young y un único ν para coeficiente de Poisson), el software requiere que el usuario tenga que hacer explícitamente el mapeo aún cuando éste es trivial. Un claro de ejemplo del sesgo "developer-easy/user-hard" que FeenoX no tiene, ya que el SRS pide que "problemas sencillos tengan inputs sencillos". En caso de que haya una única manera de asociar volúmenes geométricos a propiedades materiales, FeenoX hace la asociación implícitamente simplificando el archivo de entrada.

Según el pliego, es mandatorio que el software desarrollado sea de código abierto según la definición de la *Open Source Initiative*. El SDS (**?@sec-sds**) comienza indicando que la herramienta FeenoX es

- al software tradicional de ingeniería y
- a las bibliotecas especializadas de elementos finitos

lo que Markdown es

- a Word y
- a LaTeX

respectivamente.

Y que no sólo es de código *abierto* en el sentido de la OSI sino que también es *libre* en el sentido de la *Free Software Foundation* [1]. La diferencia entre código abierto y software libre es más sutil que práctica, ya que las definiciones técnicas prácticamente coinciden. El punto principal de que el código sea abierto es que permite obtener mejores resultados con mejor performance mientras más personas puedan estudiar el código, escrutarlo y eventualmente mejorarlo [2]. Por otro lado, el software libre persigue un fin ético relacionado con la libertad de poder ejecutar, distribuir, modificar y distribuir las modificaciones del software recibido [3], [4].

Observación. Ninguno de los dos conceptos, código abierto o software libre, se refiere a la idea de *precio*. En Español no debería haber ninguna confusión. Pero en inglés, el sustantivo adjetivado *free software* se suele tomar como gratis en lugar de libre. Si bien es cierto que la mayoría de las veces la utilización de software libre y abierto no implica el pago de ninguna tarifa directa al autor del programa, el software libre puede tener otros costos indirectos asociados.[2] El concepto importante es *libertad*: la libertad de poder contratar a uno o más programadores que modifiquen el código para que el software se comporte como uno necesita.

[2] El gerente de sistemas de una empresa de ingeniería y construcción me dijo que por un tema de costos ellos preferían usar un servidor SQL privativo antes que uno libre: "es más caro pagarle al que sabe dónde poner el punto y coma en el archivo de configuración que pagar la licencia y configurarlo con el mouse".

Tal como como Unix[3] [5] y C[4], FeenoX es un "efecto de tercer sistema"[5] [6]. De hecho, esta diferencia entre el concepto de código abierto y software libre fue discutida en la referencia [7] durante el desarrollo de la segunda versión del sistema.

De las lecciones aprendidas en las dos primeras versiones, la primera un poco naïve pero funcional y la segunda más pretenciosa y compleja (apalancada en la funcionalidad de la primera), hemos convergido al diseño explicado en el SDS del **?@sec-sds** donde definimos la filosofía de diseño del software y elegimos una de las infinitas formas de diseñar una herramienta computacional mencionadas al comienzo de este capítulo. Gran parte de este diseño está basado en la filosofía de programación Unix [6]. En el **?@sec-sds** damos ejemplos de cómo son las interfaces para definir un cierto problema y de cómo obtener los resultados. Comparamos alternativas e indicamos por qué hemos decidido diseñar el software de la forma en la que fue diseñado. Por otro lado, en este capítulo de la tesis nos centramos en la implementación, tratando de converger a una de las infinitas formas de implementar el diseño propuesto en el SDS. Comentamos muy superficialmente las ideas distintivas que tiene FeenoX con respecto a otros programas de elementos finitos desde el punto de vista técnico de programación.

Estas características son distintivas del diseño e implementación propuestos y no son comunes. En la jerga de emprendedurismo, serían las *unfair advantages* del software con respecto a otras herramientas similares.

Observación. El código fuente de FeenoX está en Github en https://github.com/seamplex/feenox/ bajo licencia GPLv3+. Consiste en aproximadamente cuarenta y cinco mil líneas de códigoorganizadas según la estructura de directorios mostrada en la figura 1.1.·

1.1. Arquitectura del código

Comencemos preguntándonos qué debemos tener en cuenta para implementar una herramienta computacional que permita resolver ecuaciones en derivadas parciales con aplicación en ingeniería utilizando el método de elementos finitos. Por el momento enfoquémonos en problemas lineales[6] como los analizados en los dos capítulos anteriores. Las dos tareas principales son

1. construir la matriz global de rigidez K y el vector $\mathbf{b}$ (o la matriz de masa M), y
2. resolver el sistema de ecuaciones $\mathsf{K} \cdot \mathbf{u} = \mathbf{b}$ (o $\mathsf{K} \cdot \mathbf{u} = \lambda \cdot \mathsf{M} \cdot \mathbf{u}$)

Haciendo énfasis en la filosofía Unix, tenemos que escribir un programa que haga bien una sola cosa[7] que nadie más hace y que interactúe con otros que h7acen bien otras cosas (regla de composición). En este sentido, nuestra herramienta se tiene que enfocar en el punto 1. Pero tenemos que definir quién va a hacer el punto 2 para que sepamos cómo es que tenemos que construir K y $\mathbf{b}$.

[3] A principios de 1960, los Bell Labs en EEUU llegaron a desarrollar un sistema operativo que funcionaba bien, así que decidieron encarar MULTICS. Como terminó siendo una monstruosidad, empezaron UNIX que es lo que quedó bien.
[4] A fines de 1960, también en los Bell Labs, llegaron a desarrollar un un lenguaje de programación A que funcionaba bien, así que decidieron encarar B. Como terminó siendo una monstruosidad, empezaron C que es lo que quedó bien.
[5] Del inglés *third-system effect*.
[6] Si el problema fuese no lineal o incluso transitorio, la discusión de esta sección sigue siendo válida para la construcción de la matriz jacobiana global.
[7] Do only one thing but do it well.

```
.
├── dist
├── doc
├── examples
├── src
│   ├── contrib
│   ├── flow
│   ├── io
│   ├── math
│   ├── mesh
│   │   └── elements
│   ├── parser
│   └── pdes
│       ├── laplace
│       ├── mechanical
│       ├── modal
│       ├── neutron_diffusion
│       ├── neutron_sn
│       └── thermal
└── tests
```

Figura 1.1.: Estructura de directorios del código fuente de FeenoX.

Las bibliotecas PETSc [8], [9] junto con la extensión SLEPc [10], [11] proveen exactamente lo que necesita una herramienta que satisfaga el SRS siguiendo la filosofía de diseño del SDS. De hecho, en 2010 seleccioné PETSc para la segunda versión del solver neutrónico por la única razón de que era una dependencia necesaria para resolver el problema de criticidad con SLEPc [12], [13]. Con el tiempo, resultó que PETSc proveía el resto de las herramientas necesarias para resolver numéricamente ecuaciones en derivadas parciales en forma portable y extensible.

Otra vez desde el punto de vista de la filosofía de programación Unix, la tarea 1 consiste en un cemento de contacto[8] entre la definición del problema a resolver por parte del ingeniero usuario y la biblioteca matemática para resolver problemas ralos[9] PETSc. Cabe preguntarnos entonces cuál es el lenguaje de programación adecuado para implementar el diseño del SDS. Aún cuando ya mencionamos que cualquier lenguaje Turing-completo es capaz de resolver un sistema de ecuaciones algebraicas, está claro que no todos son igualmente convenientes. Por ejemplo Assembly o BrainFuck son interesantes en sí mismos (por diferentes razones) pero para nada útiles para la tarea que tenemos que realizar. De la misma manera, en el otro lado de la distancia con respecto al hardware, lenguajes de alto nivel como Python también quedan fuera de la discusión por cuestiones de eficiencia computacional. A lo sumo, estos lenguajes interpretados podrían servir para proveer clientes finos[10] a través de APIs que puedan llegar a simplificar la definición del (o los) problema(s) que tenga que resolver FeenoX. Para resumir una discusión mucho más compleja, los lenguajes candidatos para implementar la herramienta requerida por el SRS podrían ser

[8] En el sentido del inglés *glue layer*.
[9] Del inglés *sparse*.
[10] Del inglés *thin clients*.

a. Fortran
b. C
c. C++

Según la regla de representación de Unix, la implementación debería poner la complejidad en las estructuras de datos más que en la lógica. Sin embargo, en el área de mecánica computacional, paradigmas de programación demasiado orientados a objetos impactan negativamente en la performance. La tendencia es encontrar un balance, tal como persigue la filosofía desde hace más de dos mil quinientos años a través de la virtud de la prudencia, entre programación orientada a objetos y programación orientada a datos.

Observación. En los últimos años el lenguaje Rust se ha comenzado a posicionar como una alternativa a C para *system programming*[11] debido al requerimiento intrínseco de que todos las referencias deben apuntar a una dirección de memoria virtual válida. A principios de 2023 aparecieron por primera vez líneas de código en Rust en el kernel de Linux. Pero desde el punto de vista de computación de alto rendimiento,[12] Rust (o incluso Go) no tienen nada nuevo que aportar con respecto a C.

Tal como explican los autores de PETSc (y coincidentemente Eric Raymond en [6]), C es el lenguaje que mejor se presta a este paradigma:

> Why is PETSc written in C, instead of Fortran or C++?
>
> When this decision was made, in the early 1990s, C enabled us to build data structures for storing sparse matrices, solver information, etc. in ways that Fortran simply did not allow. ANSI C was a complete standard that all modern C compilers supported. The language was identical on all machines. C++ was still evolving and compilers on different machines were not identical. Using C function pointers to provide data encapsulation and polyRmorphism allowed us to get many of the advantages of C++ without using such a large and more complicated language. It would have been natural and reasonable to have coded PETSc in C++; we opted to use C instead.

Fortran fue diseñado en la década de 1950 y en ese momento representó un salto cualitativo con respecto a la forma de programar las incipientes computadoras digitales [14]. Sin embargo, las suposiciones que se han tenido en cuenta con respecto al hardware y a los sistemas operativos sobre los cuales estos programas deberían ejecutarse ya no son válidas. Las revisiones posteriores como Fortran 90 son modificaciones y parches que no resuelven el problema de fondo. En cambio, C fue diseñado a principios de la década 1970 suponiendo arquitecturas de hardware y de sistemas operativos que justamente son las que se emplean hoy en día, tales como

- espacio de direcciones plano[13]
- procesadores con registros de diferentes tamaños
- llamadas al sistema[14]

[11] No hay traducción de este término. En el año 2008, se propuso una materia en el IB con este nombre en inglés. El consejo académico decidió traducirla y nombrarla como "programación de sistemas". Ni el nombre elegido ni el ligeramente más correcto "programación del sistema" tienen la misma denotación que el concepto original *system programming*

[12] Del inglés *high-performance computing* (HPC).

[13] Del inglés *flat address space.*

[14] Del inglés *system calls.*

- entrada y salida basada en archivos
- etc.

para correr en entornos Unix, que justamente, es el sistema operativo de la vasta mayoría de los servidores de la nube pública, que justa y nuevamente, es lo que perseguimos en esta tesis.

Una vez más, en principio la misma tarea puede ser implementada en cualquier lenguaje: Fortran podría implementar programación con objetos y C++ podría ser utilizado con un paradigma orientado a datos. Pero lo usual es que código escrito en Fortran sea procedural y basado en estructuras `COMMON`, resultando difícil de entender y depurar El código escrito en C++ suele ser orientado a objetos y con varias capas de encapsulamiento, polimorfismo, métodos virtuales, redefinición de operadoras y contenedores enplantillados[15] resultando difícil de entender y depurar, tal como indica Linus Torvals en la cita del comienzo del capítulo. De la misma manera que el lenguaje Fortran permite realizar ciertas prácticas que bien utilizadas agregan potencia y eficiencia pero cuyo abuso lleva a código ininteligible (por ejemplo el uso y abuso de bloques `COMMON`), C++ permite ciertas prácticas que bien utilizadas agregan potencia pero cuyo abuso lleva a código de baja calidad (por ejemplo el uso y abuso de `templates`, `shared_pointers`, `binds`, `moves`, lambdas, wrappers sobre wrappers, objetos sobre objetos, interfaz sobre interfaz, etc.), a veces en términos de eficiencia, a veces en términos del concepto de Unix *compactness* [6]:

> Compactness is the property that a design can fit inside a human being's head. A good practical test for compactness is this: Does an experienced user normally need a manual? If not, then the design (or at least the subset of it that covers normal use) is compact.

y usualmente en términos de la regla de simplicidad en la filosofía de Unix:

> Add complexity only where you must.

Justamente, Fortran y C++ hacen fácil agregar complejidad innecesaria. En C, no es fácil agregar complejidad innecesaria.

1.1.1. Construcción de los elementos globales

Habiendo decidido entonces construir la matriz `K` y el vector **b** como una glue-layer implementada en C utilizando una estructura de datos que PETSc pueda entender, preguntémonos ahora qué necesitamos para construir estos objetos. Para simplificar el argumento, supongamos por ahora que queremos resolver la ecuación generalizada de Poisson de la **?@sec-poisson**. La matriz global `K` proviene de ensamblar las matrices elementales $\mathtt{K}_i$ para todos los elementos volumétricos e_i según la **?@def-Ki-poisson**. De la misma manera, el vector global $\mathbf{b}_i$ proviene de ensamblar las contribuciones elementales $\mathbf{b}_i$ tanto de los elementos volumétricos (**?@def-bi-volumetrico-poisson**) como de los elementos de superficie con condiciones de contorno naturales (**?@def-bi-superficial-poisson**).

Una posible implementación en pseudo código de alto nivel de esta construcción podría ser la ilustrada en el algoritmo 1. La aplicación de las condiciones de contorno de Dirichlet según la discusión

[15] Del inglés *templated*.

foreach *elemento volumétrico* e_i **do**
  $\mathsf{K}_i \leftarrow 0$
  $\mathsf{b}_i \leftarrow 0$
  for *cada punto de cuadratura* $q = 1, \dots, Q_i$ **do**
    $\mathsf{J}_i(\boldsymbol{\xi}_q) \leftarrow \mathsf{B}_c(\boldsymbol{\xi}_q) \cdot \mathsf{C}_i$
    $\mathsf{B}_i(\boldsymbol{\xi}_q) \leftarrow \mathsf{J}_i^{-T}(\boldsymbol{\xi}_q) \cdot \mathsf{B}_c(\boldsymbol{\xi}_q)$
    $\mathbf{x}_q(\boldsymbol{\xi}_q) \leftarrow \sum_{j=1}^{J_i} h_j(\boldsymbol{\xi}_q) \cdot \mathbf{x}_j$
    $\mathsf{K}_i \leftarrow \mathsf{K}_i + \omega_q \cdot \left| \det \left[\mathsf{J}_i \left(\boldsymbol{\xi}_q \right) \right] \right| \left\{ \mathsf{B}_i^T(\boldsymbol{\xi}_q) \cdot k(\mathbf{x}_q) \cdot \mathsf{B}_i(\boldsymbol{\xi}_q) \right\}$
    $\mathbf{b}_i \leftarrow \mathbf{b}_i + \omega_q \cdot \left| \det \left[\mathsf{J}_i \left(\boldsymbol{\xi}_q \right) \right] \right| \left\{ \mathsf{H}_c^T(\boldsymbol{\xi}_q) \cdot f(\mathbf{x}_q) \right\}$
  end
  ensamblar $\mathsf{K}_i \to \mathsf{K}$
  ensamblar $\mathbf{b}_i \to \mathbf{b}$
end
foreach *elemento superficial* e_i *con condición de Neumann* $p(\mathbf{x})$ **do**
  $\mathsf{b}_i \leftarrow 0$
  for *cada punto de cuadratura* $q = 1, \dots, Q_i$ **do**
    $\mathbf{b}_i \leftarrow \mathbf{b}_i + \omega_q^{(D-1)} \cdot \left| \det \left[\mathsf{J}_i \left(\boldsymbol{\xi}_q \right) \right] \right| \cdot \left\{ \mathsf{H}_{c'}^T(\boldsymbol{\xi}_q) \cdot p(\mathbf{x}_q) \right\}$
  end
  ensamblar $\mathbf{b}_i \to \mathbf{b}$
end

Algoritmo 1: Posible implementación de alto nivel de la construcción de K y $\mathbf{b}$ para el problema de Poisson generalizado de la sección **??**

foreach *elemento superficial* e_i *con condición de Dirichlet* $g(\mathbf{x})$ **do**
  for *cada nodo local* $j = 1, \dots, J_i$ **do**
    calcular la fila global k correspondiente al nodo local j del elemento e_i
    hacer cero la fila k de la matriz global K
    poner un uno en la diagonal de la fila k de la matriz global K
    poner $g(\mathbf{x}_j)$ en la fila k del vector global $\mathbf{b}$
  end
end

Algoritmo 2: Una forma de poner condiciones de Dirichlet en el problema discretizado, barriendo sobre elementos y calculando la fila global a partir del nodo local.

foreach *nodo global* $j = 1, \dots J$ **do**
  foreach *elemento* e_i *al que pertenece el nodo global* j **do**
    if *el elemento* e_i *tiene una condición de Dirichlet* **then**
      hacer cero la fila j de la matriz global K
      poner un uno en la diagonal de la fila j de la matriz global K
      poner $g(\mathbf{x}_j)$ en la fila j del vector global $\mathbf{b}$
    end
  end
end

Algoritmo 3: Otra forma de poner condiciones de Dirichlet en el problema discretizado, barriendo sobre nodos globales y encontrando los elementos asociados.

de la **?@sec-dirichlet-nh** puede ser realizada con el algoritmo 2 o con el algoritmo 3. En un caso barremos sobre elementos y tenemos que encontrar el índice global asociado a cada nodo local. En otro caso barremos sobre nodos globales pero tenemos que encontrar los elementos asociados al nodo global.

La primera conclusión que podemos extraer es que para problemas escalares, la ecuación a resolver está determinada por la expresión entre llaves de los términos evaluados en cada punto de Gauss para K_i y para $\mathbf{b}_i$. Si el problema tuviese más de una incógnita por nodo y reemplazamos las matrices H_c y B_i por sus versiones G-aware H_{Gc} y B_{Gi}, entonces otra vez la ecuación a resolver está completamente definida por la expresión entre llaves.

Por ejemplo, si quisiéramos resolver difusión de neutrones multigrupo tendríamos que hacer

$$\mathsf{K}_i \leftarrow \mathsf{K}_i + \omega_q \cdot \left| \det \left[\mathsf{J}_i \left(\boldsymbol{\xi}_q \right) \right] \right| \cdot \left\{ \mathsf{L}_i(\boldsymbol{\xi}_q) + \mathsf{A}_i(\boldsymbol{\xi}_q) - \mathsf{F}_i(\boldsymbol{\xi}_q) \right\}$$

con las matrices intermedias según la **?@eq-LAF**

$$\begin{aligned} \mathsf{L}_i &= \mathsf{B}_{Gi}^T(\boldsymbol{\xi}_q) \cdot \mathsf{D}_D(\boldsymbol{\xi}_q) \cdot \mathsf{B}_{Gi}(\boldsymbol{\xi}_q) \\ \mathsf{A}_i &= \mathsf{H}_{Gc}^T(\boldsymbol{\xi}_q) \cdot \mathsf{R}(\boldsymbol{\xi}_q) \cdot \mathsf{H}_{Gc}(\boldsymbol{\xi}_q) \\ \mathsf{F}_i &= \mathsf{H}_{Gc}^T(\boldsymbol{\xi}_q) \cdot \mathsf{X}(\boldsymbol{\xi}_q) \cdot \mathsf{H}_{Gc}(\boldsymbol{\xi}_q) \end{aligned} \tag{??}$$

más las contribuciones a la matriz de rigidez para condiciones de Robin. Pero en principio podríamos escribir un algoritmo genérico (¿una cáscara?) que implemente el método de elementos finitos para resolver una cierta ecuación diferencial completamente definida por la expresión dentro de las llaves (¿pasadas como argumentos?).

La segunda conclusión proviene de preguntarnos qué es lo que necesitan los algoritmos 1, 2 y 3 para construir la matriz de rigidez global K y el vector $\mathbf{b}$:

i. los conjuntos de cuadraturas de gauss $\omega_q, \boldsymbol{\xi}_q$ para el elemento e_i
ii. las matrices canónicas H_c, B_c y H_{Gc}
iii. las matrices elementales C_i, B_{Gi},
iv. poder evaluar en $\mathbf{x}_q$

    a. las conductividad $k(\mathbf{x})$ (o las secciones eficaces para construir $\mathsf{D}_D(\boldsymbol{\xi}_q)$, $\mathsf{R}(\boldsymbol{\xi}_q)$ y $\mathsf{X}(\boldsymbol{\xi}_q)$)

b. la fuente volumétrica $f(\mathbf{x})$ (o las fuentes independientes isotrópicas $s_{0,g}(\boldsymbol{\xi}_q)$)
c. las condiciones de contorno $p(\mathbf{x})$ y $g(\mathbf{x})$ (o las correspondientes a difusión de neutrones)

Los tres primeros puntos no dependen de la ecuación a resolver. Lo único que se necesita para evaluarlos es tener disponible la topología de la malla no estructurada que discretiza el dominio $U \in \mathbb{R}^D$ a través la posición $\mathbf{x}_j \in \mathbb{R}^D$ de los nodos y la conectividad de cada uno de los elementos e_i.

El cuarto en principio sí depende de la ecuación, pero finalmente se reduce a la capacidad de evaluar

1. propiedades de materiales
2. condiciones de contorno

en una ubicación espacial arbitraria $\mathbf{x}$.

Para fijar ideas, supongamos que tenemos un problema de conducción de calor. La conductividad $k(\mathbf{x})$[16] o en general, cualquier propiedad material puede depender de la posición $\mathbf{x}$ porque

a. existen materiales con propiedades discontinuas en diferentes ubicaciones $\mathbf{x}$, y/o
b. la propiedad depende continuamente de la posición aún para el mismo material.

De la misma manera, una condición de contorno (sea esencial o natural) puede depender discontinuamente con la superficie donde esté aplicada o continuamente dentro de la misma superficie a través de alguna dependencia con la posición $\mathbf{x}$.

Entonces, la segunda conclusión es que si nuestra herramienta fuese capaz de proveer un mecanismo para definir propiedades materiales y condiciones de contorno que puedan depender

a. discontinuamente según el volúmen o superficie al que pertenezca cada elemento (algunos elementos volumétricos pertenecerán al combustible y otros al moderador, algunos elementos superficiales pertenecerán a una condición de simetría y otros a una condición de vacío) y/o
b. continuamente en el espacio según variaciones locales (por ejemplo cambios de temperatura y/o densidad, concentración de venenos, etc.)

entonces podríamos resolver ecuaciones diferenciales arbitrarias discretizadas espacialmente con el método de elementos finitos.

1.1.2. Polimorfismo con apuntadores a función

Según la discusión de la sección anterior, podemos diferenciar entre dos partes del código:

1. Una que tendrá que realizar tareas "comunes"
   - en el sentido de que son las mismas para todas las PDEs tal como leer la malla y evaluar funciones en un punto $\mathbf{x}$ arbitrario del espacio
2. Otra parte que tendrá que realizar tareas particulares para cada ecuación a resolver

[16] Si la conductividad dependiera de la temperatura T el problema sería no lineal. Pero en el paso k e la iteración de Newton tendríamos $k(\mathbf{T}_k(\mathbf{x}) = k_k(\mathbf{x})$ por lo que la discusión sigue siendo válida.

- por ejemplo evaluar las expresiones entre llaves en el q-ésimo punto de Gauss del elemento i-ésimo

Definición 1.1 (framework). Llamamos *framework* a la parte del código que implementa las tareas comunes.

Por diseño, el problema que FeenoX tiene que resolver tiene que estar completamente definido en el archivo de entrada. Entonces éste debe justamente definir qué clase de ecuación se debe resolver. Como el tipo de ecuación se lee en tiempo de ejecución, el framework debe poder ser capaz de llamar a una u otra (u otra) función que le provea la información particular que necesita: por ejemplo las expresiones entre llaves para la matriz de rigidez y para las condiciones de contorno.

Una posible implementación (ingenua) sería

if *la PDE es poisson* **then**
  evaluar $\mathsf{B}_i^T \cdot k(\mathbf{x}_q) \cdot \mathsf{B}_i$
else if *la PDE es difusión* **then**
  evaluar $\mathsf{L}_i(\boldsymbol{\xi}_q) + \mathsf{A}_i(\boldsymbol{\xi}_q) + \mathsf{F}_i(\boldsymbol{\xi}_q)$
else if *la PDE es* S_N **then**
  evaluar $\mathsf{L}_i(\boldsymbol{\xi}_q) + \mathsf{A}_i(\boldsymbol{\xi}_q) + \mathsf{F}_i(\boldsymbol{\xi}_q)$
else
  quejarse "no sé resolver esta PDE"
end

De la misma manera, necesitaríamos bloques `if` de este tipo para inicializar el problema, evaluar condiciones de contorno, calcular resultados derivados (por ejemplo flujos de calor a partir de las temperaturas, tensiones a partir de desplazamientos o corrientes a partir de flujos neutrónicos), etc. Está claro que esto es

1. feo,
2. ineficiente, y
3. difícil de extender

En C++ esto se podría implementar mediante una jerarquía de clases donde las clases hijas implementarían métodos virtuales que el framework llamaría cada vez que necesite evaluar el término entre llaves. Si bien C no tiene "métodos virtuales", sí tiene apuntadores a función (que es justamente lo que PETSc usa para implementar polimorfismocomo mencionamos en la página 6) por lo que podemos usar este mecanismo para lograr una implementación superior, que explicamos a continuación.

Por un lado, sí existe un lugar del código con un bloque `if` según el tipo de PDE requerida en tiempo de ejecución que consideramos feo, ineficiente y difícil de extender. Pero,

a. este *único* bloque de condiciones `if` se ejecutan una sola vez en el momento de analizar gramaticalmente[17] el archivo de entrada y lo que hacen es resolver un apuntador a función a la dirección de memoria de una rutina de inicialización particular que el framework debe llamar antes de comenzar a construir `K` y `b`:

[17] Del inglés *parse*.

```
if (strcasecmp(token, "laplace") == 0) {
  feenox.pde.init_parser_particular = feenox_problem_init_parser_laplace;
} else if (strcasecmp(token, "mechanical") == 0) {
  feenox.pde.init_parser_particular = feenox_problem_init_parser_mechanical;
} else if (strcasecmp(token, "modal") == 0) {
  feenox.pde.init_parser_particular = feenox_problem_init_parser_modal;
} else if (strcasecmp(token, "neutron_diffusion") == 0) {
  feenox.pde.init_parser_particular = feenox_problem_init_parser_neutron_diffusion;
} else if (strcasecmp(token, "neutron_sn") == 0) {
  feenox.pde.init_parser_particular = feenox_problem_init_parser_neutron_sn;
} else if (strcasecmp(token, "thermal") == 0) {
  feenox.pde.init_parser_particular = feenox_problem_init_parser_thermal;
} else {
  feenox_push_error_message("unknown problem type '%s'", token);
  return FEENOX_ERROR;
}
```

Estas funciones de inicialización a su vez resuelven los apuntadores a función particulares para evaluar contribuciones elementales volumétricas en puntos de Gauss, condiciones de contorno, post-procesamiento, etc.

b. El bloque `if` mostrado en el punto anterior es generado programáticamente a partir de un script (regla de Unix de generación) que analiza (*parsea*) el árbol del código fuente y, para cada subdirectorio en `src/pdes`, genera un bloque `if` automáticamente. Es fácil ver el patrón que siguen cada una de las líneas del listado en el punto a y escribir un script o macro para generarlo programáticamente.

Entonces,

1. Si bien ese bloque sigue siendo feo, es generado y compilado por una máquina que no tiene el mismo sentido estético que nosotros.

2. Reemplazamos la evaluación de n condiciones `if` para llamar a una dirección de memoria fija para cada punto de Gauss para cada elemento por una des-referencia de un apuntador a función en cada puntos de Gauss de cada elemento. En términos de eficiencia, esto es similar (tal vez más eficiente) que un método virtual de C++. Esta des-referencia dinámica no permite que el compilador pueda hacer un `inline` de la función llamada, pero el gasto extra[18] es muy pequeño. En cualquier caso, el mismo script que parsea la estructura en `src/pdes` podría modificarse para generar un binario de FeenoX para cada PDE donde en lugar de llamar a un apuntador a función se llame directamente a las funciones propiamente dichas permitiendo optimización en tiempo de vinculación[19] que le permita al compilador hacer el `inline` de la función particular (ver sección 1.3.3).

3. El script que parsea la estructura de `src/pdes` en busca de los tipos de PDEs disponibles es parte del paso `autogen.sh` (ver la discusión de la sección 1.1.4) dentro del esquema `configure` + `make` de Autotools. Las PDEs soportadas por FeenoX puede ser extendidas agregando un nuevo subdirectorio dentro de `src/pdes` donde ya existen

   - `laplace`

[18] Del inglés *overhead*.
[19] Del inglés *link-time optimization*.

- `thermal`
- `mechanical`
- `modal`
- `neutron_diffusion`
- `neutron_sn`

tomando uno de estos subdirectorios como plantilla.[20] De hecho también es posible eliminar completamente uno de estos directorios en el caso de no querer que FeenoX pueda resolver alguna PDE en particular. De esta forma, `autogen.sh` permitirá extender (o reducir) la funcionalidad del código, que es uno de los puntos solicitados en el SDS. Más aún, sería posible utilizar este mecanismo para cargar funciones particulares desde objetos compartidos[21] en tiempo de ejecución, incrementando aún más la extensibilidad de la herramienta.

1.1.3. Definiciones e instrucciones

En el **?@sec-neutronica-phwr** mencionamos (y en el **?@sec-sds** explicamos en detalle) que la herramienta desarrollada es una especie de "función de transferencia" entre uno o más archivos de entrada y cero o más archivos de salida (incluyendo la salida estándar `stdout`):

```
                           +------------+
mesh (*.msh)  }            |            |             { terminal
data (*.dat)  } input ---> |   FeenoX   |----> output { data files
input (*.fee) }            |            |             { post (vtk/msh)
                           +------------+
```

Este archivo de entrada, que a su vez puede incluir otros archivos de entrada y/o hacer referencia a otros archivos de datos (incluyendo la malla en formato `.msh`) contiene palabras clave[22] en inglés que, por decisión de diseño, deben

1. definir completamente el problema de resolver
2. ser lo más auto-descriptivas posible
3. permitir una expresión algebraica en cualquier lugar donde se necesite un valor numérico
4. mantener una correspondencia uno a uno entre la definición "humana" del problema y el archivo de entrada
5. hacer que el archivo de entrada de un problema simple sea simple

Estas keywords pueden ser

a. definiciones (sustantivos)

   - qué PDE hay que resolver
   - propiedades materiales
   - condiciones de contorno
   - variables, vectores y/o funciones auxiliares
   - etc.

[20] Del inglés *template*
[21] Del inglés *shared objects*.
[22] Del inglés *keywords*.

b. instrucciones (verbos)

- leer la malla
- resolver problema
- asignar valores a variables
- calcular integrales o encontrar extremos sobre la malla
- escribir resultados
- etc.

Las definiciones se realizan a tiempo de parseo. Una vez que el FeenoX acepta que el archivo de entrada es válido, comienza la ejecución de las instrucciones en el orden indicado en el archivo de entrada. De hecho, FeenoX tiene un apuntador a instrucción[23] que se incrementa a medida que avanza la ejecución de las instrucciones. Existen palabras clave `IF` y `WHILE` que permiten flujos de ejecución no triviales, especialmente en problemas iterativos, cuasi-estáticos o transitorios.

Por ejemplo, la lectura del archivo de malla es una instrucción y no una definición porque

i. el nombre del archivo de la malla puede depender de alguna variable cuyo valor deba ser evaluado en tiempo de ejecución con una instrucción previa. Por ejemplo

```
INPUT_FILE surprise PATH nafems-le1%g.msh round(random(0,1))
READ_MESH surprise
PRINT cells
```

```
$ feenox surprise.fee
18700
$ feenox surprise.fee
18700
$ feenox surprise.fee
32492
$ feenox surprise.fee
32492
$ feenox surprise.fee
18700
$
```

ii. el archivo con la malla en sí puede ser creado internamente por FeenoX con una instrucción previa `WRITE_MESH` y/o modificada en tiempo de ejecución

```
```feenox
READ_MESH square.msh SCALE 1e-3
WRITE_MESH square_tmp.vtk

INPUT_FILE tmp PATH square_tmp.vtk
READ_MESH tmp
```
```

iii. en problemas complejos puede ser necesario leer varias mallas antes de resolver la PDE en cuestión, por ejemplo leer una distribución de temperaturas en una malla gruesa de primer orden para utilizarla al evaluar las propiedades de los materiales de la PDE que se quiere resolver. Ver ejemplo **XXX** de non-conformal mesh mapping.

[23] Del inglés *instruction pointer*.

Comencemos con un problema sencillo para luego agregar complejidad en forma incremental. A la luz de la discusión de este capítulo, preguntémonos ahora qué necesitamos para resolver un problema de conducción de calor estacionario sobre un dominio $U\mathbb{R}^D$ con un único material:

1. definir que la PDE es conducción de calor en D dimensiones
2. leer la malla con el dominio $U \in \mathbb{R}^D$ discretizado
3. definir la conductividad térmica $k(\mathbf{x})$ del material
4. definir las condiciones de contorno del problema
5. construir `K` y $\mathbf{b}$
6. resolver $\mathtt{K} \cdot \mathbf{u} = \mathbf{b}$
7. hacer algo con los resultados
    - calcular T_{max}
    - calcular T_{mean}
    - calcular el vector flujo de calor $q''(\mathbf{x}) = -k(\mathbf{x}) \cdot \nabla T$
    - comparar con soluciones analíticas
    - etc.
8. escribir los resultados

El problema de conducción de calor más sencillo es un slab unidimensional en el intervalo $x \in [0, 1]$ con conductividad uniforme y condiciones de Dirichlet $T(0) = 0$ y $T(1) = 1$ en ambos extremos. Por diseño, el archivo de entrada tiene que ser sencillo y tener una correspondencia uno a uno con la definición "humana" del problema:

```
PROBLEM thermal 1D              # 1. definir que la PDE es calor 1D
READ_MESH slab.msh              # 2. leer la malla
k = 1                           # 3. definir conductividad uniforme igual a uno
BC left  T=0                    # 4. condiciones de contorno de Dirichlet
BC right T=1                    #    "left" y "right" son nombres en la malla
SOLVE_PROBLEM                   # 5. y 6. construir y resolver
PRINT T(0.5)                    # 7. y 8. escribir en stdout la temperatura en x=0.5
```

En este caso sencillo, la conductividad k fue dada como una variable `k` con un valor igual a uno. Estrictamente hablando, la asignación fue una instrucción. Pero en el momento de resolver el problema, las funciones particulares de la PDE de conducción de calor (dentro de `src/pdes/thermal`) buscan una variable llamada `k` y toman su valor para utilizarlo idénticamente en todos los puntos de Gauss de los elementos volumétricos al calcular el término $\mathtt{B}_i^T \cdot k \cdot \mathtt{B}_i$.

Si la conductividad no fuese uniforme sino que dependiera del espacio por ejemplo como

$$k(x) = 1 + x$$

entonces el archivo de entrada sería

```
PROBLEM thermal 1D
READ_MESH slab.msh
k(x) = 1+x                      # 3. conductividad dada por una función de x
BC left  T=0
BC right T=1
SOLVE_PROBLEM
PRINT T(1/2) log(1+1/2)/log(2)  # 7. y 8. imprimir el valor numérico y la solución analítica
```

En este caso, no hay una variable llamada `k` sino que hay una *función* de x con la expresión algebraica $1 + x$. Entonces la rutina particular dentro de `src/pde/thermal` que evalúa las contribuciones volumétricas elementales de la matriz de rigidez toma dicha función $k(x)$ como la propiedad "conductividad" y la evalúa como $\mathtt{B}_i^T(\mathbf{x}_q) \cdot k(\mathbf{x}_q) \cdot \mathtt{B}_i(\mathbf{x}_q)$. La salida, que por diseño está 100% definida por el archivo de entrada (reglas de Unix de silencio y de economía) consiste en la temperatura evaluada en $x = 1/2$ junto con la solución analítica $\log(1 + \frac{1}{2})/\log(2)$ en dicho punto.

Por completitud, mostramos que también la conductividad podría depender de la temperatura. En este caso particular el problema queda no lineal y mencionamos algunas particularidades sin ahondar en detalles. El parser algebraico de FeenoX sabe que k depende de T, por lo que la rutina particular de inicialización de la PDE de conducción de calor marca que el problema debe ser resuelto por PETSc con un objeto SNES (en lugar de un KSP como para el caso lineal). FeenoX también calcula el jacobiano necesario para resolver el problema con un método de Newton iterativo:

```
PROBLEM thermal 1D
READ_MESH slab.msh
k(x) = 1+T(x)                    # 3. la conductividad ahora depende de T(x)
BC left  T=0
BC right T=1
SOLVE_PROBLEM
PRINT T(1/2) sqrt(1+(3*0.5))-1
```

La ejecución de FeenoX sigue también las reglas tradicionales de Unix. Se debe proveer la ruta al archivo de entrada como principal argumento luego del ejecutable. Es un argumento y no como una opción ya que la funcionalidad del programa depende de que se indique un archivo de entrada, por lo que no es "opcional". Sí se pueden agregar opciones siguiendo las reglas POSIX. Algunas opciones son para FeenoX (por ejemplo `--progress` o `--elements_info` y otras son pasadas a PETSc/SLEPc (por ejemplo `--ksp_view` o `--mg_levels_pc_type=sor`). Al ejecutar los tres casos anteriores, obtenemos los resultados solicitados con la instrucción `PRINT` en la salida estándar:

```
$ feenox thermal-1d-dirichlet-uniform-k.fee
0.5
$ feenox thermal-1d-dirichlet-space-k.fee
0.584945        0.584963
$ feenox thermal-1d-dirichlet-temperature-k.fee
0.581139        0.581139
$
```

Para el caso de conducción de calor estacionario solamente hay una única propiedad cuyo nombre debe ser `k` para que las rutinas particulares la detecten como la conductividad k que debe aparecer en la contribución volumétrica. Si el problema (es decir, la malla) tuviese dos materiales diferentes, digamos `A` y `B` hay dos maneras de definir sus propiedades materiales en FeenoX:

1. agregando el sufijo `_A` y `_B` a la variable `k` o a la función `k(x)`, es decir

   ```
   k_A = 1
   k_B = 2
   ```

   si $k_A = 1$ y $k_B = 2$, o

   ```
   k_A(x) = 1+2*x
   k_B(x) = 3+4*x
   ```

si $k_A(x) = 1 + 2x$ y $k_B(x) = 3 + 4x$.

2. utilizando una palabra clave `MATERIAL` (definición) para ´cada material, del siguiente modo

```
MATERIAL A k=1
MATERIAL B k=2
```

o

```
MATERIAL A k=1+2*x
MATERIAL B k=3+4*x
```

De esta forma, el framework implementa las dos posibles dependencias de las propiedades de los materiales discutidasen la página 10:

a. continua con el espacio a través de expresiones algebraicas que pueden involucrar funciones definidas por puntos en interpoladas como discutimos en la sección 1.2.2, y
b. discontinua según el material al que pertenece el elemento.

Con respecto a las condiciones de contorno, la lógica es similar pero ligeramente más complicada. Mientras que a partir del nombre la propiedad las rutinas particulares pueden evaluar las contribuciones volumétricas, sean a la matriz de rigidez a través de la conductividad `k` o al vector **b** a través de la fuente de calor por unidad de volumen `q`, para las condiciones de contorno se necesita un poco más de información. Supongamos que una superficie tiene una condición de "simetría" y que queremos que la línea del archivo de entrada que la defina sea

```
BC left  symmetry
```

donde `left` es el nombre de la entidad superficial definida en la malla y `symmetry` es una palabra clave que indica qué tipo de condición de contorno tienen los elementos que pertenecen a `left`. La forma de implementar numéricamente (e incluso el signficado físico) esta condición de contorno depende de la PDE que estamos resolviendo. No es lo mismo poner una condición de simetría en

- poisson generalizada
- elasticidad lineal
- difusión de neutrones
- transporte por S_N

Entonces, si bien las propiedades de materiales pueden ser parseadas por el framework y aplicadas por las rutinas particulares, las condiciones de contorno deben ser parseadas y aplicadas por las rutinas particulares. De todas maneras, la lógica es similar: las rutinas particulares proveen puntos de entrada[24] que el framework llama en los momentos pertinentes durante la ejecución de las instrucciones.

Para terminar de ilustrar la idea, consideremos el problema de Reed (discutido en detalle en la sección 2.2) que propone resolver con S_N un slab uni-dimensional compuesto por varios materiales, algunos con una fuente independiente, otros sin fuente e incluso un material de vacío:

[24] Del inglés *entry points*.

```
#
#     |          |    |          |    |          |
#  m  | src= 50  | 0  |    0     | 1  |    0     |    v
#  i  |          |    |          |    |          |    a
#  r  | tot= 50  | 5  |    0     | 1  |    1     |    c
#  r  |          |    |          |    |          |    u
#  o  | scat=0   | 0  |    0     | 0.9|   0.9    |    u
#  r  |          |    |          |    |          |    m
#     |          |    |          |    |          |
#     |    1     | 2  |    3     | 4  |    5     |
#     |          |    |          |    |          |
#     +----------+----+----------+----+----------+-------> x
#    x=0        x=2  x=3        x=5  x=6        x=8

PROBLEM neutron_sn DIM 1 GROUPS 1 SN $1

READ_MESH reed.msh

MATERIAL source_abs    S1=50 Sigma_t1=50 Sigma_s1.1=0
MATERIAL absorber      S1=0  Sigma_t1=5  Sigma_s1.1=0
MATERIAL void          S1=0  Sigma_t1=0  Sigma_s1.1=0
MATERIAL source_scat   S1=1  Sigma_t1=1  Sigma_s1.1=0.9
MATERIAL reflector     S1=0  Sigma_t1=1  Sigma_s1.1=0.9

BC left  mirror
BC right vacuum

SOLVE_PROBLEM

FUNCTION ref(x) FILE reed-ref.csv INTERPOLATION steffen
PRINT sqrt(integral((ref(x)-phi1(x))^2,x,0,8))/8
```

- La primera línea es una definición (`PROBLEM` es un sustantivo) que le indica a FeenoX que debe resolver las ecuaciones S_N en $D = 1$ dimensión con $G = 1$ grupo de energías y utilizando una discretrización angular N dada por el primer argumento de la línea de comando luego del nombre del archivo de entrada. De esta manera se pueden probar diferentes discretizaciones con el mismo archivo de entrada, digamos `reed.fee`

  ```
  $ feenox reed 2   # <- S2
  [...]
  $ feenox reed 4   # <- S4
  [...]
  ```

- La segunda línea es una instrucción (el verbo `READ`) que indica que FeenoX debe leer la malla donde resolver problema del archivo `reed.msh`. Si el archivo de entrada se llamara `reed.fee`, esta línea podría haber sido `READ_MESH $0.msh`. En caso de leer varias mallas, la que define el domnio de la PDE es

  a. la primera de las instrucciones `READ_MESH`, o
  b. la definida explícitamente con la palabra clave `MAIN_MESH`

- El bloque de palabras clave `MATERIAL` definen (con un sustantivo) las secciones eficaces macroscópicas (`Sigma`) y las fuentes independientes (`S`). En este caso todas las propiedades son uniformes dentro de cada material.

- Las siguientes dos líneas definien las condiciones de contorno (`BC` quiere decir *boundary condition* que es un sustantivo adjetivado): la superficie `left` tiene una condición de simetría (o espejo) y la superficie `right` tiene una condición de vacío.
- La instrucción `SOLVE_PROBLEM` le pide a FeenoX que

  1. construya la matriz global `K` y el vector **b** (pidiéndole a su vez a las rutinas específicas del subdirectorio `src/pdes/neutron_sn`)
  2. que le pida a su vez a PETSc que resuelva el problema $\mathtt{K} \cdot \mathbf{u} = \mathbf{b}$ (como hay fuentes independientes entonces la rutina de inicialización del problema `neutron_sn` sabe que debe resolver un problema lineal, si no hubiese fuentes y sí secciones eficaces de fisión se resolvería un problema de autovalores con la biblioteca SLEPc)
  3. que extraiga los valores nodales de la solución **u** y defina las funciones del espacio $\psi_{mg}(x)$ y $\phi_g(x)$. Si `$1` fuese `2` entonces las funciones con la solución serían
     - `psi1.1(x)`
     - `psi1.2(x)`
     - `phi1(x)`

     Si `$1` fuese `4` entonces serían
     - `psi1.1(x)`
     - `psi1.2(x)`
     - `psi1.3(x)`
     - `psi1.4(x)`
     - `phi1(x)`

     Estas funciones están disponibles para que instrucciones subsiguientes las utilicen como salida directamente con ‘WRITE_MESH, como parte de otras expresiones intermedias, etc.
  4. Si el problema fuese de criticidad, entonces esta instrucción también pondría el valor del factor de multiplicación efectivo k_{eff} en una variable llamada `keff`.

- La siguiente línea define una función auxiliar de la variable espacial x a partir de un archivo de columnas de datos. Este archivo contiene una solución de referencia del problema de Reed. La función `ref(x)` puede ser evaluada en cualquier punto x utilizando una interpolación monotónica cúbica de tipo `steffen` [15].
- La última línea imprime en la salida estándar una representación ASCII (por defecto utilizando el formato `%g` de la instrucción `printf()` de C) del error L_2 cometido por la solución calculada con FeenoX con respecto a la solución de referencia, es decir

$$\frac{1}{8} \cdot \sqrt{\int_0^8 \left[\text{ref}(x) - \phi_1(x)\right]^2}$$

La ejecución de FeenoX con este archivo de entrada para S_2, S_4, S_6 y S_8 da como resultado

```
$ feenox reed.fee 2
0.0505655
$ feenox reed.fee 4
0.0143718
```

```
$ feenox reed.fee 6
0.010242
$ feenox reed.fee 8
0.0102363
$
```

1.1.4. Puntos de entrada

La compilación del código fuente usa el procedimiento recomendado por GNU donde el script `configure` genera los archivos de *make*[25] según

a. la arquitectura del hardware (Intel, ARM, etc.)
b. el sistema operativo (GNU/Linux, otras variantes, etc.)
c. las dependencias disponibles (MPI, PETSc, SLEPc, GSL, etc.)

A su vez, para generar este script `configure` se suele utilizar el conjunto de herramientas conocidas como Autotools. Estas herramientas generan, a partir de un conjunto de definiciones reducidas dadas en el lenguaje de macros M4, no sólo el script `configure` sino también otros archivos relacionados al proceso de compilación tales como los templates para los makefiles. Estas definiciones reducidas (que justamente definen las arquitecturas y sistemas operativos soportados, las dependencias, etc.) usualmente se dan en un archivo de texto llamado `configure.ac` y los templates que indican dónde están los archivos fuente que se deben compilar en archivos llamados `Makefile.am` ubicados en uno o más subdirectorios. Éstos últimos se referencian desde `configure.ac` de forma tal que Autoconf y Automake trabajen en conjunto para generar el script `configure`, que forma parte de la distribución del código fuente de forma tal que un usuario arbitrario pueda ejecutarlo y luego compilar el código con el comando `make`, que lee el `Makefile` generado por `configure`.

Para poder implementar la idea de extensibilidad según la cual FeenoX podría resolver diferentes ecuaciones en derivadas parciales, le damos una vuelta más de tuerca a esta idea de generar archivos a partir de scripts. Para ello empleamos la idea de *bootstrapping* (figura 1.2), en la cual el archivo `configure.ac` y/o las plantillas `Makefile.am` son generadas a partir de un script llamado `autogen.sh` ↩ (algunos autores prefieren llamarlo `bootstrap`).

Este script `autogen.sh` detecta qué subdirectorios hay dentro del directorio `src/pdes` y, para cada uno de ellos, agrega unas líneas a un archivo fuente llamado `src/pdes/parse.c` que hace apuntar un cierto apuntador a función a una de las funciones definidas dentro del subdirectorio. En forma resumida,

```
for pde in *; do
 if [ -d ${pde} ]; then
  if [ ${first} -eq 0 ]; then
    echo -n "  } else " >> parse.c
  else
    echo -n "  " >> parse.c
  fi
  cat << EOF >> parse.c
if (strcasecmp(token, "${pde}") == 0) {
  feenox.pde.parse_problem = feenox_problem_parse_problem_${pde};

EOF
```

[25] Del inglés *make files*.

Figura 1.2.: El concepto de `bootstrap` (también llamado `autogen.sh`).

```
  first=0
 fi
done
```

Esto generaría el bloque de `if`s feo que ya mencionamos en `parse.c`

```
  if (strcasecmp(token, "laplace") == 0) {
    feenox.pde.parse_problem = feenox_problem_parse_problem_laplace;
  } else if (strcasecmp(token, "mechanical") == 0) {
    feenox.pde.parse_problem = feenox_problem_parse_problem_mechanical;
  } else if (strcasecmp(token, "modal") == 0) {
    feenox.pde.parse_problem = feenox_problem_parse_problem_modal;
  } else if (strcasecmp(token, "neutron_diffusion") == 0) {
    feenox.pde.parse_problem = feenox_problem_parse_problem_neutron_diffusion;
  } else if (strcasecmp(token, "neutron_sn") == 0) {
    feenox.pde.parse_problem = feenox_problem_parse_problem_neutron_sn;
  } else if (strcasecmp(token, "thermal") == 0) {
    feenox.pde.parse_problem = feenox_problem_parse_problem_thermal;
  } else {
    feenox_push_error_message("unknown problem type '%s'", token);
    return FEENOX_ERROR;
  }
```

que son llamadas desde el parser general luego de haber leído la definición `PROBLEM`. Por ejemplo, si en el archivo de entrada se encuentra esta línea

```
PROBLEM laplace
```

entonces el apuntador a función `feenox.pde.parse_problem` declarado en `feenox.h` como

```
int (*parse_problem)(const char *token);
```

apuntaría a la función `feenox_problem_parse_problem_laplace()` declarada en `src/pdes/laplace/methods_h` y definida en `src/pdes/laplace/parser.c`. De hecho, esta función es llamada con el parámetro `token`

conteniendo cada una de las palabras que está a continuación del nombre del problema que el parser general no entienda. En particular, para la línea

```
PROBLEM neutron_sn DIM 3 GROUPS 2 SN 8
```

lo que sucede en tiempo de parseo es

1. La palabra clave primaria `PROBLEM` es leída (y entendida) por el parser general.
2. El argumento `neutron_sn` es leído por el bloque generado por `autogen.sh`. El apuntador a la función global `feenox.pde.parse_problem` se hace apuntar entonces a `feenox_problem_parse_problem_neutron_sn()`.
3. La palabra clave secundaria `DIM` es leída por el parser general. Como es una palabra clave secundaria asociada a la primaria `PROBLEM` y el parser general la entiende (ya que el framework tiene que saber la dimensión espacial de la ecuación diferencial en derivadas parciales que tiene que resolver). Entonces lee el siguiente argumento `3` y sabe que tiene que resolver una ecuación diferencial sobre tres dimensiones espaciales. Esto implica, por ejemplo, definir que las propiedades de los materiales y las soluciones serán funciones de tres argumentos: x, y y z.
4. La palabra clave secundaria `GROUPS` es leída por el parser general. Como no es una palabra clave secundaria asociada a la primara `PROBLEM`, entonces se llama a `feenox.pde.parse_problem` (que apunta a `feenox_problem_parse_problem_neutron_sn()`) con el argumento `token` apuntando a `GROUPS`.
5. Como el parser particular dentro de `src/pdes/neutron_sn` sí entiende que la palabra clave `GROUPS` define la cantidad de grupos de energía, lee el siguiente token `2`, lo entiende como tal y lo alamacena en la estructura de datos particular de las rutinas correspondientes a `neutron_sn`.
6. El control vuelve al parser principal que lee la siguiente palabra clave secundaria `SN`. Como tampoco la entiende, vuelve a llamar al parser particular que entiende que debe utilizar las direcciones y pesos de S_8.
7. Una vez más el control vuelve al parser principal, que llega al final de la línea. En este momento, vuelve a llamar al parser específico `feenox_problem_parse_problem_neutron_sn()` pero pasando `NULL` como argumento. En este punto, se considera que el parser específico ya tiene toda la información necesaria para inicializar (al menos una parte) de sus estructuras internas y de las variables o funciones que deben estar disponibles para otras palabras claves genéricas. Por ejemplo, si el problema es neutrónico entonces inmediatamente después de haber parseado completamente la línea `PROBLEM` debe definirse la variable `keff` y las funciones con los flujos escalares (y angulares si correspondiere) de forma tal que las siguientes líneas, que serán interpretadas por el parser genérico, entiendan que `keff` es una variable y que `phi1(x,y,z)`, `psi1.1(x,y,z)` y `psi8.2(x,y,z)` son expresiones válidas:

```
PRINT "keff = " keff
PRINT " rho = " (1-keff)/keff
PRINT psi1.1(0,0,0) psi8.2(0,0,0)
profile(x) = phi1(x,x,0)
PRINT_FUNCTION profile phi1(x,0,0) MIN 0 MAX 20 NSTEPS 100
```

Dentro de las inicializaciones en tiempo de parseo, cada implementación específica debe resolver el resto de los apuntadores a función que definen los puntos de entrada específicos que el framework principal necesita llamar para

i. parsear partes específicas del archivo de entrada

   a. condiciones de contorno
   b. la palabra clave `WRITE_RESULTS` que escribe "automáticamente" los resultados en un archivo de post-procesamiento en formato `.msh` o `.vtk`. Esto es necesario ya que las rutinas que escriben los resultados son parte del framework general pero dependiendo de la PDE a resolver e incluso de los detalles de la PDE (por ejemplo la cantidad de grupos de energía en un problema neutrónico o la cantidad de modos calculadas en un problema de análsis de modos naturales de oscilación mecánicos).

ii. inicializar estructuras internas

iii. solicitar la memoria virtual necesaria al sistema operativo[26] y construir las matrices y los vectores globales

iv. resolver las ecuaciones discretizadas con PETSc (o SLEPc) según el tipo de problema resultante:

   a. problema lineal en estado estacionario $\mathsf{K} \cdot \mathbf{u} = \mathbf{b}$ (PETSc KSP)
   b. problema generalizado de autovalores $\mathsf{K} \cdot \mathbf{u} = \lambda \cdot \mathsf{M} \cdot \mathbf{u}$ (SLEPc EPS)
   c. problema no lineal en estado estacinario $\mathbf{F}(\mathbf{u}) = 0$ (PETSc SNES)
   d. problema transitorio $\mathbf{G}(\mathbf{u}, \dot{\mathbf{u}}, t) = 0$ (PETSc TS)

v. calcular campos secundarios a partir de los primarios (ver sección 1.1.4.5), por ejemplo

   - flujos escalares a partir de flujos angulares
   - corrientes a partir de flujos neutrónicos
   - flujos de calor a partir de temperaturas
   - tensiones a partir de deformaciones
   - etc.

```c
// parse
int (*parse_problem)(const char *token);
int (*parse_write_results)(mesh_write_t *mesh_write, const char *token);
int (*parse_bc)(bc_data_t *bc_data, const char *lhs, char *rhs);

// init
int (*init_before_run)(void);
int (*setup_pc)(PC pc);
int (*setup_ksp)(KSP ksp);
int (*setup_eps)(EPS eps);
int (*setup_ts)(TS ksp);

// build·
int (*element_build_volumetric)(element_t *e);
int (*element_build_volumetric_at_gauss)(element_t *e, unsigned int q);

// solve
int (*solve)(void);
```

[26] Del inglés *allocate*.

```
// post
int (*solve_post)(void);
int (*gradient_fill)(void);
int (*gradient_nodal_properties)(element_t *e, mesh_t *mesh);
int (*gradient_alloc_nodal_fluxes)(node_t *node);
int (*gradient_add_elemental_contribution_to_node)(node_t *node, element_t *e, unsigned int
    j, double rel_weight);
int (*gradient_fill_fluxes)(mesh_t *mesh, size_t j_global);
```

1.1.4.1. Parseo

Cuando se termina la línea de `PROBLEM`, el parser general llama a `parse_problem(NULL)` que debe

1. terminar de rellenar los apuntadores específicos

```
// virtual methods
feenox.pde.parse_bc = feenox_problem_bc_parse_neutron_diffusion;
feenox.pde.parse_write_results = feenox_problem_parse_write_post_neutron_diffusion;

feenox.pde.init_before_run = feenox_problem_init_runtime_neutron_diffusion;

feenox.pde.setup_eps = feenox_problem_setup_eps_neutron_diffusion;
feenox.pde.setup_ksp = feenox_problem_setup_ksp_neutron_diffusion;
feenox.pde.setup_pc = feenox_problem_setup_pc_neutron_diffusion;

feenox.pde.element_build_volumetric = feenox_problem_build_volumetric_neutron_diffusion;
feenox.pde.element_build_volumetric_at_gauss =
    feenox_problem_build_volumetric_gauss_point_neutron_diffusion;

feenox.pde.solve_post = feenox_problem_solve_post_neutron_diffusion;
```

2. inicializar lo que necesita el parser para poder continuar leyendo el problema específico, incluyendo

   - la definición de variables especiales (por ejemplo los flujos escalares `phi_1`, `phi_2`, etc. y angulares `psi_1.1`, `psi_2.1`, ..., `psi_12.2` y las variables `keff` y `sn_alpha`) para que estén disponibes para el parser algebraico (ver sección 1.2.1)

```
// the angular fluxes psi
feenox_check_alloc(feenox.pde.unknown_name = calloc(feenox.pde.dofs, sizeof(char
    *)));
for (unsigned int m = 0; m < neutron_sn.directions; m++) {
  for (unsigned int g = 0; g < neutron_sn.groups; g++) {
    feenox_check_minusone(asprintf(&feenox.pde.unknown_name[m * neutron_sn.groups
        + g], "psi%u.%u", m+1, g+1));
  }
}

// the scalar fluxes psi
feenox_check_alloc(neutron_sn.phi = calloc(neutron_sn.groups, sizeof(function_t *)));
for (unsigned int g = 0; g < neutron_sn.groups; g++) {
  char *name = NULL;
  feenox_check_minusone(asprintf(&name, "phi%u", g+1));
  feenox_call(feenox_problem_define_solution_function(name, &neutron_sn.phi[g],
      FEENOX_SOLUTION_NOT_GRADIENT));
```

```
  feenox_free(name);
}

neutron_sn.keff = feenox_define_variable_get_ptr("keff");

neutron_sn.sn_alpha = feenox_define_variable_get_ptr("sn_alpha");
feenox_var_value(neutron_sn.sn_alpha) = 0.5;
```

- el seteo de opciones por defecto (¿qué pasa si no hay keywords `GROUPS` o `SN`?)

```
// default is 1 group
if (neutron_sn.groups == 0) {
  neutron_sn.groups = 1;
}

// default is N=2
if (neutron_sn.N == 0) {
  neutron_sn.N = 2;
}
if (neutron_sn.N % 2 != 0) {
  feenox_push_error_message("number of ordinates N = %d has to be even",
      neutron_sn.N);
  return FEENOX_ERROR;
}

// stammler's eq 6.19
switch(feenox.pde.dim) {
  case 1:
    neutron_sn.directions = neutron_sn.N;
    break;
  case 2:
    neutron_sn.directions = 0.5*neutron_sn.N*(neutron_sn.N+2);
    break;
  case 3:
    neutron_sn.directions = neutron_sn.N*(neutron_sn.N+2);
    break;
}

// dofs = number of directions * number of groups
feenox.pde.dofs =  neutron_sn.directions * neutron_sn.groups;
```

- la inicialización de direcciones $\hat{\mathbf{\Omega}}_m$ y pesos w_m para S_N

Cuando una línea contiene la palabra clave principial `BC` tal como

```
BC left  mirror
BC right vacuum
```

entonces el parser principal lee el token siguiente, `left` y `right` respectivamente, que intentará vincular con un grupo físico en la malla del problema. Los siguientes tokens, en este caso sólo uno para cada caso, son pasados al parser específico `feenox.pde.parse_bc` que sabe qué hacer con las palabras `mirror` y `vacuum`. Este parser de condiciones de contorno a su vez resuelve uno de los dos apuntadores

```
int (*set_essential)(bc_data_t *, element_t *, size_t j_global);
int (*set_natural)(bc_data_t *, element_t *, unsigned int q);
```

dentro de la estructura asociada a la condición de contorno según corresponda al tipo de condición.

1.1.4.2. Inicialización

Luego de que el parser lee completamente el archivo de entrada, FeenoX ejecuta las instrucciones en el orden adecuado posiblemente siguiendo lazos de control y bucles[27]. Al llegar a la instrucción `SOLVE_PROBLEM`, llama al punto de entrada `init_before_run()` donde

1. Se leen las expresiones que definen las propiedades de los materiales, que no estaban disponibles en el momento de la primera inicialización después de leer la línea `PROBLEM`. Estas propiedades de materiales en general y las secciones eficaces macroscópicas en particular pueden estar dadas por

   a. variables
   b. funciones del espacio
   c. la palabra clave `MATERIAL`

   En este momento de la ejecución todas las secciones eficaces deben estar definidas con nombres especiales. Por ejemplo,

   - Σ_{tg} = `"Sigma_t%d"`
   - Σ_{ag} = `"Sigma_a%d"`
   - $\nu\Sigma_{fg}$ = `"nuSigma_f%d"`
   - $\Sigma_{s_0 g\to g'}$ = `"Sigma_s%d.%d"`
   - $\Sigma_{s_1 g\to g'}$ = `"Sigma_s_one%d.%d"`
   - S_g = `"S%d"`

2. Dependiendo de si hay fuentes de fisión y/o fuentes independientes se determina si hay que resolver un problema lineal o un problema de autovalores generalizado. En el primer caso se hace

   ```
   feenox.pde.math_type = math_type_linear;
   feenox.pde.solve     = feenox_problem_solve_petsc_linear;
   feenox.pde.has_mass  = 0;
   feenox.pde.has_rhs   = 1;
   ```

   y en el segundo

   ```
   feenox.pde.math_type = math_type_eigen;
   feenox.pde.solve     = feenox_problem_solve_slepc_eigen;
   feenox.pde.has_mass  = 1;
   feenox.pde.has_rhs   = 0;
   ```

Luego de esta segunda inicialización, se llama al apuntador `feenox.pde.solve` que para neutrónica es o bien

a. `feenox_problem_solve_petsc_linear()` que construye K y $\mathbf{b}$ y resuelve $\mathsf{K}\cdot\mathbf{u}=\mathbf{b}$, o
b. `feenox_problem_solve_slepc_eigen()` que construye K y M y resuelve $\mathsf{K}\cdot\mathbf{u}=\lambda\cdot\mathsf{M}\cdot\mathbf{u}$.

[27] Del inglés *loop*

Antes de construir y resolver las ecuaciones, se llama a su vez a los apuntadores a función que correspondan

- `feenox_pde.setup_pc(PC pc)`
- `feenox.pde.setup_ksp(KSP kps)`
- `feenox.pde.setup_eps(EPS eps)`

donde cada problema particular configura el precondicionador, el solver lineal y el solver de autovalores en caso de que el usuario no haya elegido algoritmos explícitamente en el archivo de entrada. Si el operador diferencial es elíptico y simétrico (por ejemplo para conducción de calor o elasticidad lineal) tal vez convenga usar por defecto un solver iterativo basado en gradientes conjugados precondicionado con multi-grilla geométrica-algebraica[28] [16]. En cambio para un operador hiperbólico no simétrico (por ejemplo S_N multigrupo) es necesario un solver más robusto como LU.

1.1.4.3. Construcción

Para construir las matrices globales `K` y/o `M` y/o el vector global $\mathbf{b}$ el framework general hace un loop sobre todos los elementos volumétricos (locales a cada proceso) y, para cada uno de ellos, primero llama al punto de entrada `feenox.pde.element_build_volumetric()` que toma un apuntador al elemento como único argumento. Cada elemento es una estructura tipo `element_t` definida en `feenox.h` como

```
struct element_t {
  size_t index;
  size_t tag;

  double quality;
  double volume;
  double area;
  double size;
  double gradient_weight;   // this weight is used to average the contribution of this ↩
      element to nodal gradients
  double *w;                // weights of the gauss points time determinant of the jacobian
  double **x;               // coordinates fo the gauss points
  double *normal;           // outward normal direction (only for 2d elements)

  // matrix with the coordinates (to compute the jacobian)
  gsl_matrix *C;

  element_type_t *type;                 // pointer to the element type
  physical_group_t *physical_group;     // pointer to the physical group this element ↩
      belongs to
  node_t **node;                        // pointer to the nodes, node[j] points to the j-th ↩
      local node
  cell_t *cell;                         // pointer to the associated cell (only for FVM)
};
```

Luego el framework general hace un sub-bucle sobre el índice q de los puntos de Gauss y llama a `feenox.pde.element_build_volumetric_at_gauss()` que toma el apuntador al elemento y el índice q como argumentos:

[28] Del inglés *geometric algeraic multi-grid*.

```
// if the specific pde wants, it can loop over gauss point in the call
// or it can just do some things that are per-element only and then loop below
if (feenox.pde.element_build_volumetric) {
  feenox_call(feenox.pde.element_build_volumetric(this));
}

// or, if there's an entry point for gauss points, then we do the loop here
if (feenox.pde.element_build_volumetric_at_gauss != NULL) {
  int Q = this->type->gauss[feenox.pde.mesh->integration].Q;
  for (unsigned int q = 0; q < Q; q++) {
    feenox_call(feenox.pde.element_build_volumetric_at_gauss(this, q));
  }
}
```

Estos dos métodos son responsables de devolver los objetos elementales volumétricos K_i, M_i como matrices densas en formato de la GNU Scientific Library `gsl_matrix` y el vector elemental $\mathbf{b}_i$ como `gsl_vector` almacenados en los apuntadores globales dentro de la estructura `feenox.fem`:

```
// elemental (local) objects
gsl_matrix *Ki;               // elementary stiffness matrix
gsl_matrix *Mi;               // elementary mass matrix
gsl_matrix *JKi;              // elementary jacobian for stiffness matrix
gsl_matrix *Jbi;              // elementary jacobian for RHS vector
gsl_vector *bi;               // elementary right-hand side vector
```

Observación. Las matrices jacobianas son necesarias para problemas no lineales resueltos con SNES.

Para el caso de difusión de neutrones, las dos funciones que construyen los objetos elementales son

```
int feenox_problem_build_volumetric_neutron_diffusion(element_t *e) {
  if (neutron_diffusion.space_XS == 0) {
    feenox_call(feenox_problem_neutron_diffusion_eval_XS(feenox_fem_get_material(e), NULL));
  }
  return FEENOX_OK;
}

int feenox_problem_build_volumetric_gauss_point_neutron_diffusion(element_t *e, unsigned int ↩
    q) {
  if (neutron_diffusion.space_XS != 0) {
    double *x = feenox_fem_compute_x_at_gauss(e, q, feenox.pde.mesh->integration);
    feenox_call(feenox_problem_neutron_diffusion_eval_XS(feenox_fem_get_material(e), x));
  }

  // elemental stiffness for the diffusion term B'*D*B
  double wdet = feenox_fem_compute_w_det_at_gauss(e, q, feenox.pde.mesh->integration);
  gsl_matrix *B = feenox_fem_compute_B_G_at_gauss(e, q, feenox.pde.mesh->integration);
  feenox_call(feenox_blas_BtCB(B, neutron_diffusion.D_G, neutron_diffusion.DB, wdet, ↩
      neutron_diffusion.Li));

  // elemental scattering H'*A*H
  gsl_matrix *H = feenox_fem_compute_H_Gc_at_gauss(e, q, feenox.pde.mesh->integration);
  feenox_call(feenox_blas_BtCB(H, neutron_diffusion.R, neutron_diffusion.RH, wdet, ↩
      neutron_diffusion.Ai));

  // elemental fission matrix
  if (neutron_diffusion.has_fission) {
```

```
    feenox_call(feenox_blas_BtCB(H, neutron_diffusion.X, neutron_diffusion.XH, wdet,  ↩
        neutron_diffusion.Fi));
  }

  if (neutron_diffusion.has_sources) {
    feenox_call(feenox_blas_Atb_accum(H, neutron_diffusion.s, wdet, feenox.fem.bi));
  }

  // for source-driven problems
  //   Ki = Li + Ai - Xi
  // for criticallity problems
  //   Ki = Li + Ai
  //   Mi = Xi
  feenox_call(gsl_matrix_add(neutron_diffusion.Li, neutron_diffusion.Ai));
  if (neutron_diffusion.has_fission) {
    if (neutron_diffusion.has_sources) {
      feenox_call(gsl_matrix_scale(neutron_diffusion.Fi, -1.0));
      feenox_call(gsl_matrix_add(neutron_diffusion.Li, neutron_diffusion.Fi));
    } else {
      feenox_call(gsl_matrix_add(feenox.fem.Mi, neutron_diffusion.Fi));
    }
  }
  feenox_call(gsl_matrix_add(feenox.fem.Ki, neutron_diffusion.Li));

  return FEENOX_OK;
}
```

Luego de hacer el loop sobre cada punto de Gauss q de cada elemento volumétrico e_i, el framework general se encarga de ensamblar los objetos globales K, M y/o $\mathbf{b}$ en formato PETSc a partir de los objetos locales K_i, M_i y/o $\mathbf{b}_i$ en formato GSL. En forma análoga, el framework hace un loop sobre los elementos superficiales que contienen condiciones de borde naturales y llama al apuntador a función `set_natural()` que toma como argumentos

1. un apuntador a una estructura `bc_data_t` definida como

```
struct bc_data_t {
  char *string;

  enum {
    bc_type_math_undefined,
    bc_type_math_dirichlet,
    bc_type_math_neumann,
    bc_type_math_robin,
    bc_type_math_multifreedom
  } type_math;

  int type_phys;      // problem-based flag that tells which type of BC this is

  // boolean flags
  int space_dependent;
  int nonlinear;
  int disabled;
  int fills_matrix;

  unsigned int dof;  // -1 means "all" dofs
  expr_t expr;
  expr_t condition;  // if it is not null the BC only applies if this evaluates to non-zero

  int (*set_essential)(bc_data_t *, element_t *, size_t j_global);
```

```
  int (*set_natural)(bc_data_t *, element_t *, unsigned int q);

  bc_data_t *prev, *next;   // doubly-linked list in ech bc_t
};
```

que es "rellenada" por el parser específico de condiciones de contorno,

2. un apuntador al elemento
3. el índice q del punto de Gauss

Por ejemplo, la condición de contorno natural tipo `vacuum` de difusión de neutrones es

```
int feenox_problem_bc_set_neutron_diffusion_vacuum(bc_data_t *this, element_t *e, unsigned
    int q) {

  feenox_fem_compute_x_at_gauss_if_needed_and_update_var(e, q, feenox.pde.mesh->integration,
      this->space_dependent);
  double coeff = (this->expr.items != NULL) ? feenox_expression_eval(&this->expr) : 0.5;

  double wdet = feenox_fem_compute_w_det_at_gauss(e, q, feenox.pde.mesh->integration);
  gsl_matrix *H = feenox_fem_compute_H_Gc_at_gauss(e, q, feenox.pde.mesh->integration);
  feenox_call(feenox_blas_BtB_accum(H, wdet*coeff, feenox.fem.Ki));

  return FEENOX_OK;
}
```

Observación. La condición de contorno tipo `mirror` en difusión, tal como en la ecuación de Laplace o en conducción de calor, consiste en "no hacer nada".

Las condiciones de contorno esenciales se ponen en el problema luego de haber ensamblado la matriz global `A` y transformarla en la matriz de rigidez `K` según el procedimiento discutido en la **?@sec-fem**. Para ello debemos hacer un loop sobre los nodos, bien con el algoritmo 2 o con el algoritmo 3, y poner un uno en la diagonal de la matriz de rigidez y el valor de la condición de contorno en el nodo en el vector del miembro derecho, en la fila correspondiente al grado de libertad global.

Observación. En un problema de autovalores, sólo es posible poner condiciones de contorno homogéneas. En este caso, se puede

a. poner un uno en la diagonal de `K` y un cero en la diagonal de `M`, ó
b. poner un uno en la diagonal de `M` y un cero en la diagonal de `K`

Cuál de las dos opciones es la más conveniente depende del algoritmo seleccionado para la transformación espectral del problema de autovalores a resolver.

La forma de implementar esto en FeenoX para una PDE arbitraria es que para cada nodo que contiene una condición de Dirichlet, el framework llama al apuntador a función `set_essential()` que toma como argumentos

1. un apuntador a la estructura `bc_data_t`
2. un apuntador al elemento (estructura `element_t`)
3. el índice j global del nodo

Para difusión, la condición `null` se implementa sencillamente como

```
int feenox_problem_bc_set_neutron_diffusion_null(bc_data_t *this, element_t *e, size_t ↩
    j_global) {
  for (unsigned int g = 0; g < feenox.pde.dofs; g++) {
    feenox_call(feenox_problem_dirichlet_add(feenox.pde.mesh->node[j_global].index_dof[g], 0));
  }
  return FEENOX_OK;
}
```

En S_N, la condición `vacuum` es ligeramente más compleja ya que debemos verificar que la dirección sea entrante:

```
int feenox_problem_bc_set_neutron_sn_vacuum(bc_data_t *this, element_t *e, size_t j_global) {

  double outward_normal[3];
  feenox_call(feenox_mesh_compute_outward_normal(e, outward_normal));
  for (unsigned m = 0; m < neutron_sn.directions; m++) {
    if (feenox_mesh_dot(neutron_sn.Omega[m], outward_normal) < 0) {
      // if the direction is inward set it to zero
      for (unsigned int g = 0; g < neutron_sn.groups; g++) {
        feenox_call(feenox_problem_dirichlet_add(
         feenox.pde.mesh->node[j_global].index_dof[sn_dof_index(m,g)], 0)
        );
      }
    }
  }

  return FEENOX_OK;
}
```

La condición de contorno `mirror` es más compleja aún porque implica condiciones multi-libertad[29], que deben ser puestas en la matriz de rigidez usando

1. Eliminación directa
2. Método de penalidad
3. Multiplicadores de Lagrange

En su versión actual, FeenoX utiliza el método de penalidad [17]. El framework provee una llamada genérica `feenox_problem_multifreedom_add()` donde las rutinas particulares deben "registrar" sus condiciones multi-libertad:

```
int feenox_problem_bc_set_neutron_sn_mirror(bc_data_t *this, element_t *e, size_t j_global) {

  double outward_normal[3];
  double reflected[3];
  double Omega_dot_outward = 0;
  double eps = feenox_var_value(feenox.mesh.vars.eps);

  feenox_call(feenox_mesh_compute_outward_normal(e, outward_normal));
  for (unsigned m = 0; m < neutron_sn.directions; m++) {
    if ((Omega_dot_outward = feenox_mesh_dot(neutron_sn.Omega[m], outward_normal)) < 0) {
      // if the direction is inward then we have to reflect it
      // if Omega is the incident direction with respect to the outward normal then
      // reflected = Omega - 2*(Omega dot outward_normal) * outward_normal
```

[29] Del inglés *multi-freedom*.

```
      for (int d = 0; d < 3; d++) {
        reflected[d] = neutron_sn.Omega[m][d] - 2*Omega_dot_outward * outward_normal[d];
      }

      unsigned int m_prime = 0;
      for (m_prime = 0; m_prime < neutron_sn.directions; m_prime++) {
        if (fabs(reflected[0] - neutron_sn.Omega[m_prime][0]) < eps &&
            fabs(reflected[1] - neutron_sn.Omega[m_prime][1]) < eps &&
            fabs(reflected[2] - neutron_sn.Omega[m_prime][2]) < eps) {
          break;
        }
      }

      if (m_prime == neutron_sn.directions) {
        feenox_push_error_message("cannot find a reflected direction for m=%d (out of %d in S%d) for node %d", m, neutron_sn.directions, neutron_sn.N, feenox.pde.mesh->node[j_global].tag);
        return FEENOX_ERROR;
      }

      double *coefficients = NULL;
      feenox_check_alloc(coefficients = calloc(feenox.pde.dofs, sizeof(double)));

      for (unsigned int g = 0; g < neutron_sn.groups; g++) {
        coefficients[sn_dof_index(m,g)] = -1;
        coefficients[sn_dof_index(m_prime,g)] = +1;
      }
      feenox_call(feenox_problem_multifreedom_add(j_global, coefficients));
      feenox_free(coefficients);
    }
  }

  return FEENOX_OK;
}
```

1.1.4.4. Solución

Una vez construidos los objetos globales `K` y/o `M` y/o **b**, el framework llama a `feenox.pde.solve()` que puede apuntar a

1. `feenox_problem_solve_petsc_linear()`
2. `feenox_problem_solve_petsc_nonlinear()`
3. `feenox_problem_solve_slepc_eigen()`
4. `feenox_problem_solve_petsc_transient()`

según el inicializador particular de la PDE haya elegido. Por ejemplo, si la variable `end_time` es diferente de cero, entonces en `thermal` se llama a `trasient()`. De otra manera, si la conductividad k depende de la temperatura T, se llama a `nonlinear()` y si no depende de T, se llama a `linear()`. En el caso de neutrónica, tanto difusión como S_N, la lógica depende de si existen fuentes independientes o no:

```
feenox.pde.math_type = neutron_diffusion.has_sources ? math_type_linear : math_type_eigen;
feenox.pde.solve     = neutron_diffusion.has_sources ? feenox_problem_solve_petsc_linear : feenox_problem_solve_slepc_eigen;

feenox.pde.has_stiffness = 1;
```

```
feenox.pde.has_mass = !neutron_diffusion.has_sources;
feenox.pde.has_rhs  =  neutron_diffusion.has_sources;
```

1.1.4.5. Post-procesamiento

Luego de resolver el problema discretizado y encontrar el vector global solución **u**, debemos dejar disponibles la solución para que las instrucciones que siguen a `SOLVE_PROBLEM` tales como `PRINT` ↩ o `WRITE_RESULTS` puedan operar sobre ellas.

Definición 1.2 (campos principales). Las funciones del espacio que son solución de la ecuación a resolver y cuyos valores nodales provienen de los elementos del vector solución **u** se llaman *campos principales*.

Definición 1.3 (campos secundarios). Las funciones del espacio que dan información sobre la solución del problema cuyos valores nodales provienen de operar algebraica, diferencial o integralmente sobre el vector solución **u** se llaman *campos secundarios*.

Problema	Campo principal	Campos secundarios
Conducción de calor	Temperatura	Flujos de calor
Elasticidad (formulación u)	Desplazamientos	Tensiones y deformaciones
Difusión de neutrones	Flujos escalares	Corrientes
Ordenadas discretas	Flujos angulares	Flujos escalares y corrientes

Cuadro 1.1.: Campos principales y secundarios.

La tabla 1.1 lista los campos principales y secundarios de algunos tipos de problemas físicos. Los campos principales son "rellenados" por el framework general mientras que los campos secundarios necesitan más puntos de entrada específicos para poder ser definidos.

En efecto, el framework sabe cómo "rellenar" las funciones solución a partir de **u** mediante los índices que mapean los grados de libertad de cada nodo espacial con los índices globales. En efecto, como ya vimos, cada inicializador debe "registrar" la cantidad y el nombre de las soluciones según la cantidad de grados de libertad por nodo dado en `feenox.pde.dofs`.

Por ejemplo, para un problema escalar como `thermal`, hay un único grado de libertad por nodo por lo que debe haber una única función solución de la ecuación que el inicializador "registró" en el framework como

```
// thermal is a scalar problem
feenox.pde.dofs = 1;
feenox_check_alloc(feenox.pde.unknown_name = calloc(feenox.pde.dofs, sizeof(char *)));
feenox_check_alloc(feenox.pde.unknown_name[0] = strdup("T"));
```

En difusión de neutrones, la cantidad de grados de libertad por nodo es la cantidad G de grupos de energía, leído por el parser específico en la palabra clave `GROUPS`. Las funciones solución son `phi1`, `phi2`, etc:

```
// default is 1 group
if (neutron_diffusion.groups == 0) {
  neutron_diffusion.groups = 1;
}
// dofs = number of groups
feenox.pde.dofs = neutron_diffusion.groups;

feenox_check_alloc(feenox.pde.unknown_name = calloc(neutron_diffusion.groups, sizeof(char *)));
for (unsigned int g = 0; g < neutron_diffusion.groups; g++) {
  feenox_check_minusone(asprintf(&feenox.pde.unknown_name[g], "phi%u", g+1));
}
```

En S_N, es el producto entre M y G. Las funciones son `psi1.1`, `psi1.2`, etc. donde el primer índice es m y el segundo es g:

```
// dofs = number of directions * number of groups
feenox.pde.dofs =  neutron_sn.directions * neutron_sn.groups;

// the angular fluxes psi
feenox_check_alloc(feenox.pde.unknown_name = calloc(feenox.pde.dofs, sizeof(char *)));
for (unsigned int m = 0; m < neutron_sn.directions; m++) {
  for (unsigned int g = 0; g < neutron_sn.groups; g++) {
    feenox_check_minusone(asprintf(&feenox.pde.unknown_name[m * neutron_sn.groups + g],  ↩
        "psi%u.%u", m+1, g+1));
  }
}
```

TODO flujos escalares y corrientes en Sn

TODO corrientes en difusión

1.2. Algoritmos auxiliares

1.2.1. Expresiones algebraicas

Una característica distintiva de FeenoX es que en cada lugar del archivo de entrada donde se espere un valor numérico, desde la cantidad de grupos de energía después de la palabra clave `GROUPS` hasta las propiedades de los materiales, es posible escribir una expresión algebraica. Por ejemplo

```
PROBLEM neutron_diffusion DIMENSIONS 1+2 GROUPS sqrt(4)
MATERIAL fuel nuSigma_f1=1+T(x,y,z)  nuSigma_f2=10-1e-2*(T(x,y,z)-T0)^2
```

Esto obedece a una de las primeras decisiones de diseño del código. Parafraseando la idea de Unix "todo es un archivo", para FeenoX "todo es una expresión". La explicación se basa en la historia misma de por qué en algún momento de mi vida profesional es que decidí escribir la primera versión de este código, cuya tercera implementación es FeenoX **?@sec-history**. Luego de la tesis de grado [18] y de la de maestría [19], en el año 2009 busqué en StackOverflow cómo implementar un parser PEMDAS (*Parenthesis Exponents Multiplication Division Addition Subtraction*) en C. Esa primera implementación solamente soportaba constantes numéricas, por lo que luego agregué la posibilidad de incorporar variables y funciones matemáticas estándar (trigonométricas, exponenciales, etc.) cuyos

argumentos son, a su vez, expresiones algebraicas. Finalmente funciones definidas por el usuario (sección 1.2.2) e incluso funcionales que devuelven un número real a partir de una expresión (implementadas con la GNU Scientific Library) como por ejemplo

- derivación

```
VAR t'
f'(t) = derivative(f(t'),t',t)
```

- integración

```
# this integral is equal to 22/7-pi
piapprox = 22/7-integral((x^4*(1-x)^4)/(1+x^2), x, 0, 1)
```

- sumatoria

```
# the abraham sharp sum (twenty-one terms)
piapprox = sum(2*(-1)^i * 3^(1/2-i)/(2*i+1), i, 0, 20)
```

- búsqueda de extremos en un dimensión

```
PRINT %.7f func_min(cos(x)+1,x,0,6)
```

- búsqueda de raíces en una dimensión

```
VECTOR kl[5]
kl[i] = root(cosh(t)*cos(t)+1, t, 3*i-2,3*i+1)
```

La forma en la que FeenoX maneja expresiones es la siguiente:

1. El parser toma una cadena de caracteres y la analiza (*parsea*, por eso es un *parser*) para generar un árbol de sintaxis abstracto[30] que consiste en una estructura `expr_t`

```
// algebraic expression
struct expr_t {
  expr_item_t *items;
  double value;
  char *string;     // just in case we keep the original string

  // lists of which variables and functions this expression depends on
  var_ll_t *variables;
  function_ll_t *functions;

  expr_t *next;
};
```

   que tiene una lista simplemente vinculada[31] de estructuras `expr_item_t`

[30] Del inglés *abstract syntax tree*.
[31] Del inglés *single-linked list*.

```
// individual item (factor) of an algebraic expression
struct expr_item_t {
  size_t n_chars;
  int type;           // defines #EXPR_ because we need to operate with masks

  size_t level;       // hierarchical level
  size_t tmp_level;   // for partial sums

  size_t oper;        // number of the operator if applicable
  double constant;    // value of the numerical constant if applicable
  double value;       // current value

  // vector with (optional) auxiliary stuff (last value, integral accumulator, rng, etc)
  double *aux;

  builtin_function_t *builtin_function;
  builtin_vectorfunction_t *builtin_vectorfunction;
  builtin_functional_t *builtin_functional;

  var_t *variable;
  vector_t *vector;
  matrix_t *matrix;
  function_t *function;

  vector_t **vector_arg;

  var_t *functional_var_arg;

  // algebraic expression of the arguments of the function
  expr_t *arg;

  // lists of which variables and functions this item (and its daughters)
  var_ll_t *variables;
  function_ll_t *functions;


  expr_item_t *next;
};
```

que pueden ser

a. un operador algebraico (`+`, `-`, `*`, `/` o `^`)
b. una constante numérica (`12.34` o `1.234e1`)
c. una variable como por (`t` o `x`)
d. un elemento de un vector (`v[1]` o `p[1+i]`)
e. un elemento de una matriz (`A(1,1)` o `B(1+i,1+j)`)
f. una función interna como por ejemplo (`sin(w*pi*t)` o `exp(-x)` o `atan2(y,x)`)
g. un funcional interno como por ejemplo (`integral(x^2,x,0,1)` o `root(cos(x)-x,x,0,1)`)
h. una función definida por el usuario (`f(x,y,z)` o `g(t)`)

2. Cada vez que se necesita el valor numérico de la expresión, se recorre la lista `items` evaluando cada uno de los factores según su orden de procedencia jerárquica PEMDAS.

Observación. Algunas expresiones como por ejemplo la cantidad de grupos de energía deben poder evaluarse en tiempo de parseo. Si bien FeenoX permite introducir expresiones en el archivo de en-

trada, éstas no deben depender de variables o de funciones ya que éstas necesitan estar en modo ejecución para tener valores diferentes de cero.

Observación. Los índices de elementos de vectores o matrices y los argumentos de funciones y funcionales son expresiones, que deben ser evaluadas en forma recursiva al recorrer la lista de ítems de una expresión base.

Observación. Los argumentos en la línea de comando `$1`, `$2`, etc. se reemplazan como si fuesen cadenas antes de parsear la expresión.

Esta funcionalidad, entre otras cosas, permite comparar resultados numéricos con resultados analíticos. Como muchas veces estas soluciones analíticas están dadas por series de potencias, el funcional `sum()` es muy útil para realizar esta comparación. Por ejemplo, en conducción de calor transitoria:

```
# example of a 1D heat transient problem
# from https://www.math.ubc.ca/~peirce/M257_316_2012_Lecture_20.pdf
# T(0,t) = 0       for t < 1
#          A*(t-1) for t > 1
# T(L,t) = 0
# T(x,0) = 0
READ_MESH slab-1d-0.1m.msh DIMENSIONS 1
PROBLEM thermal

end_time = 2

# unitary non-dimensional properties
k = 1
rhocp = 1
alpha = k/rhocp

# initial condition
T_0(x) = 0
# analytical solution
# example 20.2 equation 20.25
A = 1.23456789
L = 0.1
N = 100
T_a(x,t) = A*(t-1)*(1-x/L) + 2*A*L^2/(pi^3*alpha^2) *  ↩
    sum((exp(-alpha^2*(i*pi/L)^2*(t-1))-1)/i^3 * sin(i*pi*x/L), i, 1, N)

# boundary conditions
BC left  T=if(t>1,A*(t-1),0)
BC right T=0

SOLVE_PROBLEM

IF t>1 # the analytical solution overflows t<1
 PRINT %.7f t T(0.5*L) T_a(0.5*L,t) T(0.5*L)-T_a(0.5*L,t)
ENDIF
```

```
$ feenox thermal-slab-transient.fee
1.0018730       0.0006274       0.0005100       0.0001174
1.0047112       0.0021005       0.0021442       -0.0000436
1.0072771       0.0036783       0.0037210       -0.0000427
1.0097632       0.0052402       0.0052551       -0.0000150
1.0131721       0.0073607       0.0073593       0.0000014
1.0192879       0.0111393       0.0111345       0.0000048
```

```
1.0315195       0.0186881       0.0186849       0.0000032
1.0500875       0.0301491       0.0301466       0.0000025
1.0872233       0.0530725       0.0530700       0.0000025
1.1614950       0.0989193       0.0989167       0.0000026
1.3100385       0.1906127       0.1906102       0.0000026
1.6071253       0.3739997       0.3739971       0.0000026
2.0000000       0.6165149       0.6165123       0.0000026
$
```

1.2.2. Evaluación de funciones

Tal como las expresiones de la sección anterior, el concepto de *funciones* es central para FeenoX como oposición y negación definitiva de la idea de "tabla" para dar dependencias no triviales de las secciones eficaces con respecto a

i. temperaturas
ii. quemados
iii. concentración de venenos
iv. etc.

según lo discutido en la referencia [7] sobre la segunda versión del código.

Una función está completamente definida por un nombre único f, por la cantidad n de argumentos que toma y por la forma de evaluar el número real $f(\mathbf{x}) : \mathbb{R}^n \mapsto \mathbb{R}$ que corresponde a los argumentos $\mathbf{x} \in \mathbb{R}^n$. Las funciones en FeenoX pueden ser

1. algebraicas, o
2. definidas por puntos

1.2.2.1. Funciones definidas algebraicamente

En el caso de funciones algebraicas, los argumentos tienen que ser variables que luego aparecen en la expresión que define la función. El valor de la función proviene de asignar a las variables de los argumentos los valores numéricos de la invocación y luego evaluar la expresión algebraica que la define. Por ejemplo

```
f(x) = 1/2*x^2
PRINT f(1) f(2)
```

Al evaluar `f(1)` FeenoX pone el valor de la variable `x` igual a 1 y luego evalúa la expresión `1/2*x^2` dando como resultado `0.5`. En la segunda evaluación, `x` vale 2 y la función se evalúa como `2`.

Si en la expresión aparecen otras variables que no forman parte de los argumentos, la evaluación de la función dependerá del valor que tengan estas variables (que se toman como parámetros) al momento de la invocación. Por ejemplo,

```
VAR v0
v(x0,t) = x0 + v0*t
PRINT v(1)
v0 = 1
PRINT v(1)
```

en la primera evaluación obtendremos `0` y en la segunda `1`.

Observación. La **?@fig-harmonics** fue creada por la herramienta de post-procesamiento tridimensinoal Paraview a partir de un archivo VTK generado por FeenoX con el siguiente archivo de entrada:

```
READ_MESH sphere.msh

Y00(x,y,z) = sqrt(1/(4*pi))

Y1m1(x,y,z) = sqrt(3/(4*pi)) * y
Y10(x,y,z)  = sqrt(3/(4*pi)) * z
Y1p1(x,y,z) = sqrt(3/(4*pi)) * x

Y2m2(x,y,z)  = sqrt(15/(4*pi)) * x*y
Y2m1(x,y,z)  = sqrt(15/(4*pi)) * y*z
Y20(x,y,z)   = sqrt(5/(16*pi)) * (-x^2-y^2+2*z^2)
Y2p1(x,y,z)  = sqrt(15/(4*pi)) * z*x
Y2p2(x,y,z)  = sqrt(15/(16*pi)) * (x^2-y^2)

WRITE_MESH harmonics.vtk Y00 Y1m1 Y10 Y1p1 Y2m2 Y2m1 Y20 Y2p1 Y2p2
```

1.2.2.2. Funciones definidas por puntos sin topología

Por otro lado, las funciones definidas por puntos pueden ser uni-dimensionales o multi-dimensionales. Las multi-dimensionales pueden tener o no topología. Y todas las funciones definidas por puntos pueden provenir de datos

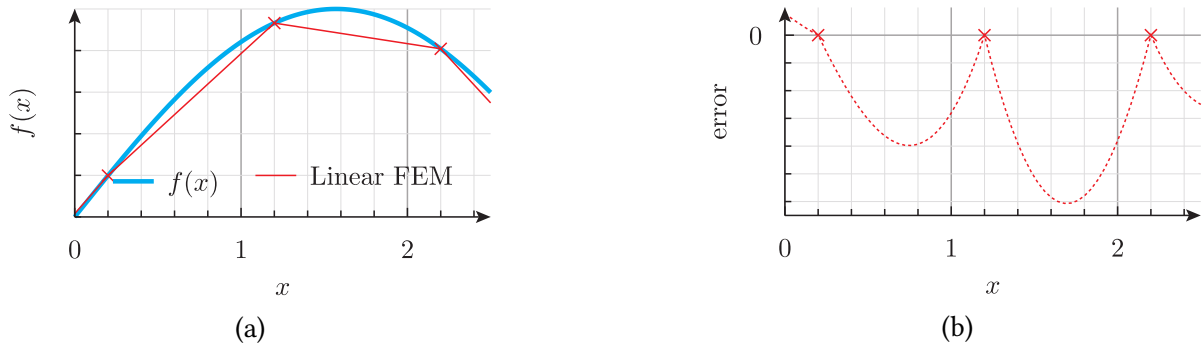
a. dentro del archivo de entrada
b. de otro archivo de datos por columnas
c. de archivos de mallas (`.msh` o `.vtk`)
d. de vectores de FeenoX (posiblemente modificados en tiempo de ejecución)

De hecho aunque no provengan estrictamente de vectores (podrían hacerlo con la palabra clave `FUNCTION VECTOR`), FeenoX provee acceso a los vectores que contienen tanto los valores independientes (puntos de definición) como los valores dependientes (valores que toma la función). En la página 41 ilustramos este acceso.

Las funciones definidas por puntos que dependen de un único argumento tienen siempre una topología implícita. FeenoX utiliza el framework de interpolación unidimensional `gsl_interp` de la GNU Scientific Library. En principio, en este tipo de funciones el nombre de los argumentos no es importante. Pero el siguiente ejemplo ilustra que no es lo mismo definir `f(x)` que `f(y)` ya que de otra manera la instrucción `PRINT_FUNCTION` daría idénticamente `sin(y)`, con el valor que tuviera la variable `y` al momento de ejecutar la instrucción `PRINT_FUNCTION` en lugar del perfil `sin(x)` que es lo que se busca:

```
FUNCTION f(x) DATA {
0.2 sin(0.2)
0.4 sin(0.4)
0.6 sin(0.6)
0.8 sin(0.8)
}

PRINT_FUNCTION f sin(x) MIN 0.2 MAX 0.8 NSTEPS 100
```

Este pequeño archivo de entrada—que además muestra que una función definida por puntos puede usar expresiones algebraicas—fue usado para generar la figura 1.3.



(a) (b)

Figura 1.3.: Dos figuras para ilustrar que el error cometido en una aproximación lineal de elementos finitos es mayor lejos de los nodos fabricada a partir de datos numéricos generados por FeenoX.

Como mencionamos, las funciones definidas por puntos de varias dimensiones pueden tener o no una topología asociada. Si no la tienen, la forma más naïve de interpolar una función de k argumentos $f(\mathbf{x})$ con $\mathbf{x} \in \mathbb{R}^k$ de estas características es asignar al punto de evaluación $\mathbf{x} \in \mathbb{R}^k$ el valor f_i del punto de definición $\mathbf{x}_i$ más cercano a $\mathbf{x}$.

Observación. La determinación de cuál es el punto de definición $\mathbf{x}_i$ más cercano a $\mathbf{x}$ se realiza en orden $O(\log N)$ con un árbol k-d[32] conteniendo todos los puntos de definición $\mathbf{x}_i \in \mathbb{R}^k$ para $i = 1, \ldots, N$. La implementación del k-d tree no es parte de FeenoX sino una biblioteca externa libre y abierta.

Observación. La noción de "punto más cercano" involucra una métrica del espacio de definición $\mathbb{R}^k$. Si las k componentes tienen las mismas unidades, se puede emplear la distancia euclideana usual. Pero por ejemplo si una componente es una temperatura y otra una presión, la métrica euclideana depende de las unidades en la que se expresan las componentes por lo que deja de ser apropiada.

Una segunda forma de evaluar la función es con una interpolación tipo Shepard [20], original o modificada. La primera consiste en realizar una suma pesada con alguna potencia p de la distancia del punto de evaluación $\mathbf{x}$ a todos los N puntos de definición de la función

$$f(\mathbf{x}) = \frac{\sum_{i=1}^{N} w_i(\mathbf{x}) \cdot f_i}{w_i(\mathbf{x})}$$

[32]Del inglés *k-dimensional tree.*

donde

$$w_i(\mathbf{x}) = \frac{1}{|\mathbf{x} - \mathbf{x}_i|^p}$$

La versión modificada consiste en sumar solamente las contribuciones correspondientes a los puntos de definición que se encuentren dentro de una hiper-bola de radio R alrededor del punto de evaluación $\mathbf{x} \in \mathbb{R}^k$.

Observación. La determinación de qué puntos $\mathbf{x}_i$ están dentro de la hiper-bola de centro $\mathbf{x}$ y de radio R también se realiza con un árbol k-d.

1.2.2.3. Funciones definidas por puntos con topología implícita

Si los puntos de definición están en una grilla multidimensional estructurada rectangularmente (no necesariamente con incrementos uniformes), entonces FeenoX puede detectar la topología implícita y realizar una interpolación local a partir de los vértices del hiper-cubo que contiene el punto de evaluación $\mathbf{x} \in \mathbb{R}^n$. Esta interpolación local es similar a la explicada a continuación para el caso de topología explícita mediante una generalización de las funciones de forma para los elementos producto-tensor de primer orden a una dimensión arbitraria k.

Por ejemplo, si se tiene el siguiente archivo con tres columnas

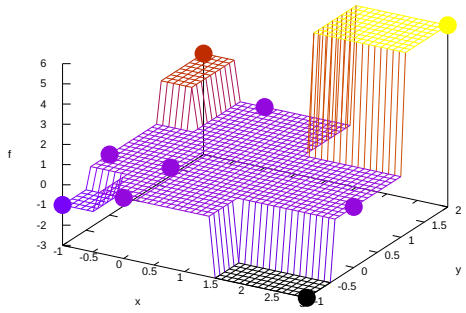
1. x_i
2. y_i
3. $f_i = f(x_i, y_i)$

```
-1 -1   -1
-1  0    0
-1 +2    2
 0 -1    0
 0  0    0
 0 +2    0
+3 -1   -3
+3  0    0
+3 +2   +6
```

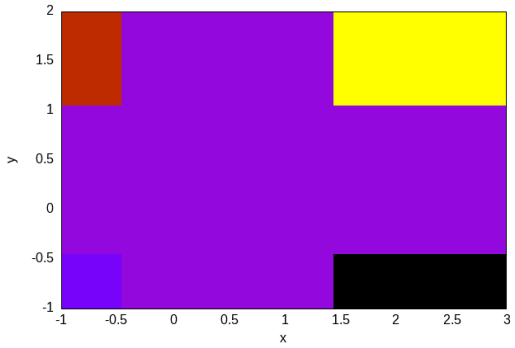
donde no hay una topología explícita pero sí una rectangular implícita, entonces podemos comparar las tres interpolaciones con el archivo de entrada

```
FUNCTION f(x,y) FILE hyperbolic-paraboloid.dat INTERPOLATION nearest
FUNCTION g(x,y) FILE hyperbolic-paraboloid.dat INTERPOLATION shepard
FUNCTION h(x,y) FILE hyperbolic-paraboloid.dat INTERPOLATION rectangular

PRINT_FUNCTION f g h MIN -1 -1 MAX 3 2 NSTEPS 40 30
```

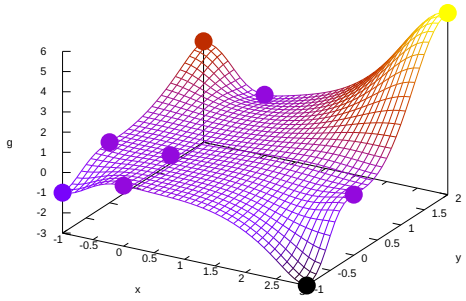
para obtener la figura 1.4.



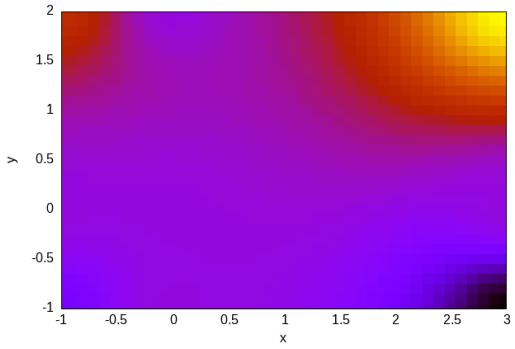
(a) $f(x, y)$ (`nearest`)



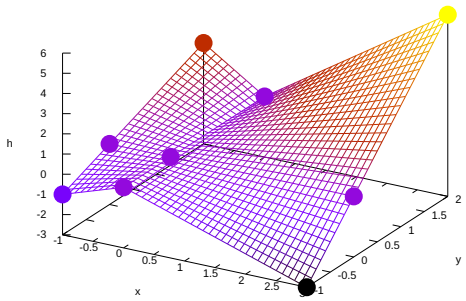
(b) $f(x, y)$ (`nearest`)



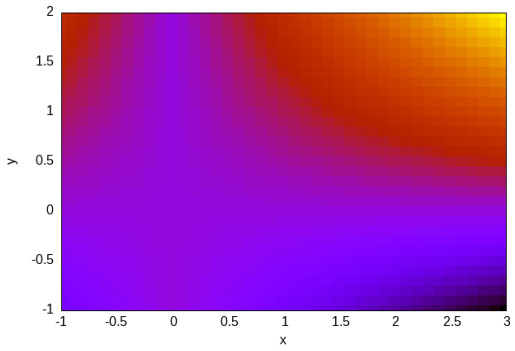
(c) $g(x, y)$ (`shepard`)



(d) $g(x, y)$ (`shepard`)



(e) $h(x, y)$ (`rectangular`)



(f) $h(x, y)$ (`rectangular`)

Figura 1.4.: Tres formas de interpolar funciones definidas por puntos a partir del mismo conjunto de datos con topología implícita.

1.2.2.4. Funciones definidas por puntos con topología explícita

Otra forma de definir y evaluar funciones definidas por puntos es cuando existe una topología explícita. Esto es, cuando los puntos de definición forman parte de una malla no estructurada con una conectividad conocida. En este caso, dada una función $f(\mathbf{x})$, el procedimiento para evaluarla en $\mathbf{x} \in \mathbb{R}^2$ o $\mathbf{x} \in \mathbb{R}^3$ es el siguiente:

1. Encontrar el elemento e_i que contiene al punto $\mathbf{x}$
2. Encontrar las coordenadas locales $\boldsymbol{\xi}$ del punto $\mathbf{x}$ en e_i
3. Evaluar las J funciones de forma $h_j(\boldsymbol{\xi})$ del elemento e_i en el punto $\boldsymbol{\xi}$
4. Calcular $f(\mathbf{x})$ a partir de los J valores nodales de defínción f_j como

$$f(\mathbf{x}) = \sum_{j=1}^{J} h_j(\boldsymbol{\xi}) \cdot f_j$$

La forma particular de implementar los puntos 1 y 2 (especialmente el 1) es crucial en términos de perfomance. FeenoX busca el elemento e_i con una combinación de un k-d tree para encontrar el nodo más cercano al punto $\mathbf{x}$ y una lista de elementos asociados a cada nodo. Una vez encontrado el elemento e_i, resolvemos un sistema de ecuaciones de tamaño J para encontrar las coordenadas locales $\boldsymbol{\xi}$.

De esta manera, si en lugar de tener los puntos de definición completamente estructuradode la página 41 tuviésemos la misma información pero en lugar de incluir el punto de definición $f(0,0) = 0$ tuviésemos $f(0.5, 0.5) = 0.25$ pero con la topología asociada (figura 1.5).

```
READ_MESH 2d-interpolation-topology.msh DIM 2 READ_FUNCTION f
PRINT_FUNCTION f MIN -1 -1 MAX 3 2 NSTEPS 40 30
```

Esta funcionalidad permite realizar lo que se conoce como "mapeo de mallas no conformes". Es decir, utilizar una distribución de alguna propiedad espacial (digamos la temperatura) dada por valores nodales en una cierta malla (de un solver térmico) para evaluar propiedades de materiales (digamos la sección eficaz de fisión) en la malla de cálculo. En este caso en particular, los puntos $\mathbf{x}$ donde se requiere evaluar la función definida en la otra malla de definición corresponden a los puntos de Gauss de los elementos de la malla de cálculo. En el caso de la sección 2.1 profundizamos este concepto.

Observación. Es importante remarcar que para todas las funciones definidas por puntos, FeenoX utiliza un esquema de memoria en la cual los datos numéricos tanto de la ubicación de los puntos de definición $\mathbf{x}_j$ como de los valores f_j de definición están disponibles para lectura y/o escritura como vectores accesibles como expresiones en tiempo de ejecución. Esto quiere decir que esta herramienta puede leer la posición de los nodos de un archivo de malla fijo y los valores de definición de alguna otra fuente que provea un vector del tamaño adecuado, como por ejemplo un recurso de memoria compartida o un socket TCP. Por ejemplo, podemos modificar el valor de definición $f(0.5, 0.5) = 0.25$ a $f(0.5, 0.5) = 40$ en tiempo de ejecución como

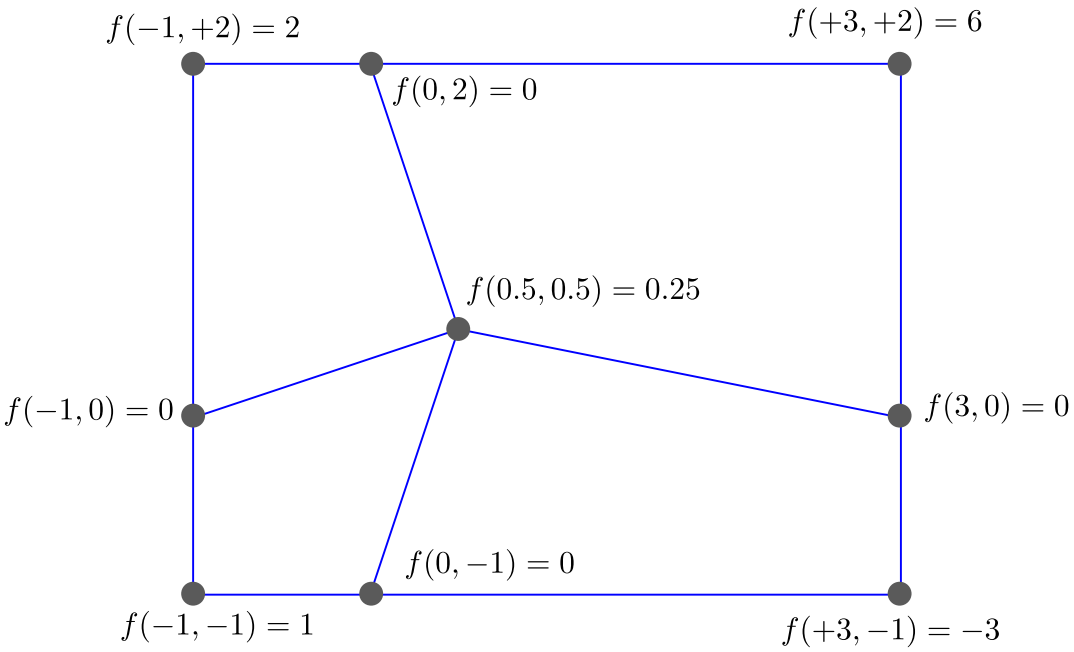
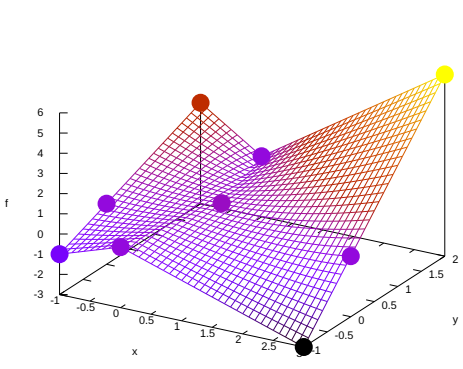
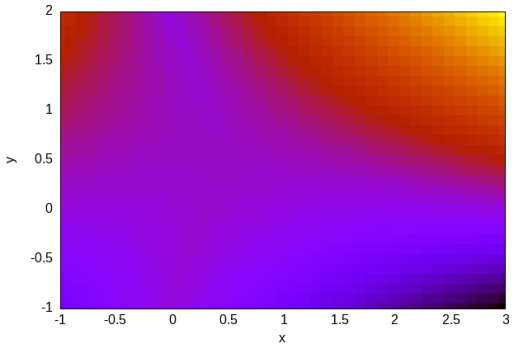


Figura 1.5.: Definición de una función $f(x,y)$ a partir de 9 datos discretos f_j y con una topología bi-dimensional explícita.



(a) $f(x,y)$ con topología

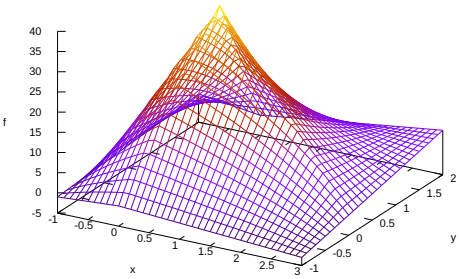


(b) $f(x,y)$ con topología

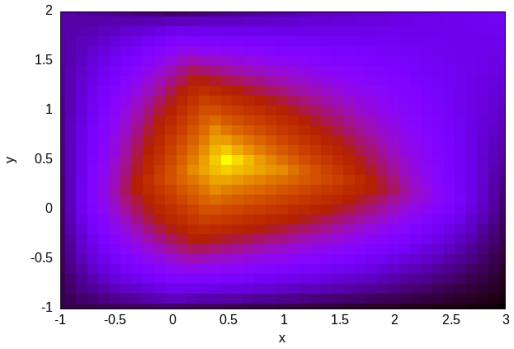
Figura 1.6.: Interpolación de una función definida por puntos con una topología explícita. La función interpolada coincide con la $h(x,y,)$ de la figura 1.4.

```
READ_MESH 2d-interpolation-topology.msh DIM 2 READ_FUNCTION f

# modificamos el valor de f(0.5,0.5) de 0.25 a 40
vec_f[5] = 40

PRINT_FUNCTION f MIN -1 -1 MAX 3 2 NSTEPS 40 30
```

para obtener la figura 1.7.



(a) $f(x, y)$ con topología y datos modificados



(b) $f(x, y)$ con topología y datos modificados

Figura 1.7.: Ilustración de la modificación en tiempo de ejecución de los datos de definición de una función definida con puntos (en este caso, con topología explícita).

1.3. Otros aspectos

1.3.1. Filosofía Unix

?@sec-unix

[6] [5]

- Regla de composición
- Regla de simplicidad
- Regla de parsimonia
- Regla de transparencia
- Regla de generación
- Regla de diversidad

Git

Bucles paramétricos a través de la línea de comandos

1.3.2. Simulación programática

No me gusta el término "simulación" pero es como se llama en la industria.

- interfaz en lenguaje de alto nivel
- entrada definida 100% en un archivo ASCII

Symbios

1.3.3. Performance

Balance entre CPU y memoria

Hay un montón de cosas para hacer!

Comparar con LE10, sparselizard, etc.

Hay un repositorio para medir con google-benchmark

Investigar y medir

- efecto de inlining
- eficiencia de acceso a memoria (cache misses)
- data-oriented design
- dmplex para topología
- have compile-time macros that will optimize for
  - speed
  - memory
  - something else
- create FeenoX flavors with compile-time
  - problem type (so we can avoid function pointers)
  - problem dimension (so we can hardcode sizes)
  - full or axi-symmetry
  - scalar size (float or double)
  - all elements are of the same type

Idea: el sistema de templates de C++ es Turing complete. Se podría hacer un solver (con la malla y las propiedades embebidas) que corra en tiempo 0 pero que tarde muchísimo en compilar.

replace with compile-time macros

enable LTO and measure

GPU con PETSc

BLAS, MKL

repositorio https://github.com/seamplex/feenox-benchmark

1.3.4. Escalabilidad

Paralelización

- MPI, no openmp
- hay un montón para hacer
- Metis: gmsh o dmplex

```
mpiexec --verbose --oversubscribe --hostfile hosts -np 4 ./feenox hello_mpi.fee
```

La instrucción `PRINTF_ALL` hace que todos los procesos escriban en la salida estándar los datos formateados con los especificadores de `printf` las variables indicadas, prefijando cada línea con la identificación del proceso y el nombre del *host*.

```
ubuntu@ip-172-31-44-208:~/mpi/hello$ mpiexec --verbose --oversubscribe --hostfile hosts -np 4 ./ ↩
    feenox hello_mpi.fee
[0/4 ip-172-31-44-208] Hello MPI World!
[1/4 ip-172-31-44-208] Hello MPI World!
[2/4 ip-172-31-34-195] Hello MPI World!
[3/4 ip-172-31-34-195] Hello MPI World!
ubuntu@ip-172-31-44-208:~/mpi/hello$
```

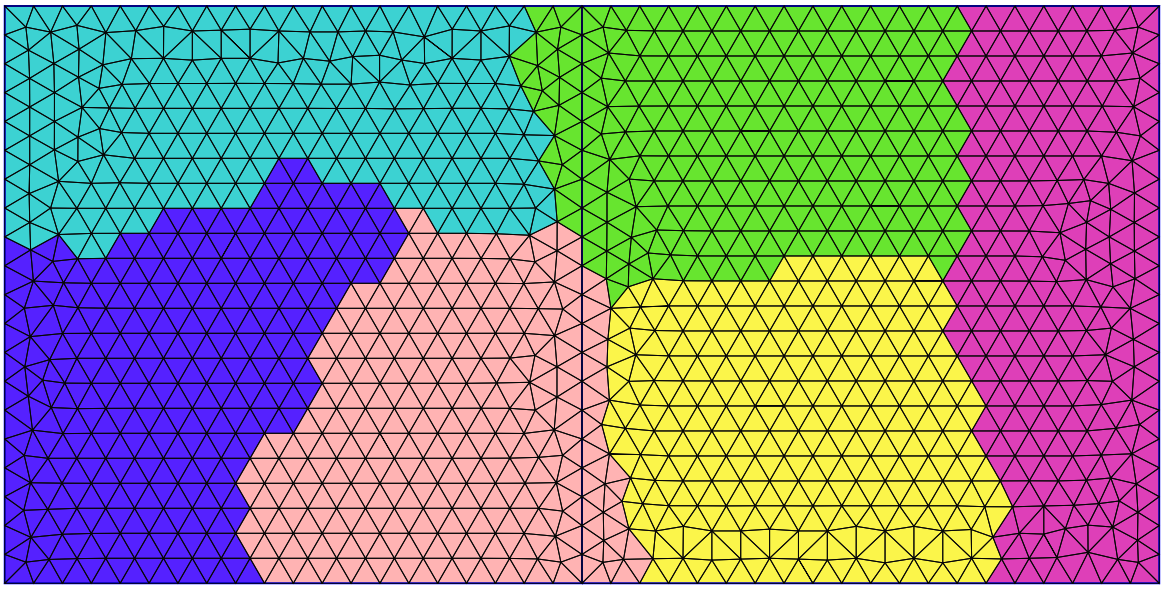


Figura 1.8.: Geometría tutorial `t21` de Gmsh: dos cuadrados mallados con triángulos y descompuestos en 6 particiones.

Podemos utilizar el tutorial `t21` de Gmsh en el que se ilustra el concepto de DDM (descomposición de dominio o particionado de la malla[33]) para mostrar cómo funciona la paralelización por MPI en FeenoX. En efecto, consideremos la malla de la figura 1.8 que consiste en dos cuadrados adimensionales de tamaño 1×1 y supongamos queremos integrar la constante 1 sobre la superficie para obtener como resultado el valor número 2.

[33]Del inglés *mesh partitioning*.

```
READ_MESH t21.msh
INTEGRATE 1 RESULT two
PRINTF_ALL "%g" two
```

En este caso, la instrucción `INTEGRATE` se calcula en paralelo donde cada proceso calcula una integral local y antes de pasar a la siguiente instrucción, todos los procesos hacen una operación de reducción mediante la cual se suman todas las contribuciones y todos los procesos obtienen el valor global en la variable `two`:

```
$ mpiexec -n 2 feenox t21.fee
[0/2 tom] 2
[1/2 tom] 2
$ mpiexec -n 4 feenox t21.fee
[0/4 tom] 2
[1/4 tom] 2
[2/4 tom] 2
[3/4 tom] 2
$ mpiexec -n 6 feenox t21.fee
[0/6 tom] 2
[1/6 tom] 2
[2/6 tom] 2
[3/6 tom] 2
[4/6 tom] 2
[5/6 tom] 2
$
```

Para ver lo que efectivamente está pasando, modificamos temporalmente el código fuente de FeenoX para que cada proceso muestre la sumatoria parcial:

```
$ mpiexec -n 2 feenox t21.fee
[proceso 0] mi integral parcial es 0.996699
[proceso 1] mi integral parcial es 1.0033
[0/2 tom] 2
[1/2 tom] 2
$ mpiexec -n 3 feenox t21.fee
[proceso 0] mi integral parcial es 0.658438
[proceso 1] mi integral parcial es 0.672813
[proceso 2] mi integral parcial es 0.668749
[0/3 tom] 2
[1/3 tom] 2
[2/3 tom] 2
$ mpiexec -n 4 feenox t21.fee
[proceso 0] mi integral parcial es 0.505285
[proceso 1] mi integral parcial es 0.496811
[proceso 2] mi integral parcial es 0.500788
[proceso 3] mi integral parcial es 0.497116
[0/4 tom] 2
[1/4 tom] 2
[2/4 tom] 2
[3/4 tom] 2
$ mpiexec -n 5 feenox t21.fee
[proceso 0] mi integral parcial es 0.403677
[proceso 1] mi integral parcial es 0.401883
[proceso 2] mi integral parcial es 0.399116
```

```
[proceso 3] mi integral parcial es 0.400042
[proceso 4] mi integral parcial es 0.395281
[0/5 tom] 2
[1/5 tom] 2
[2/5 tom] 2
[3/5 tom] 2
[4/5 tom] 2
$ mpiexec -n 6 feenox t21.fee
[proceso 0] mi integral parcial es 0.327539
[proceso 1] mi integral parcial es 0.330899
[proceso 2] mi integral parcial es 0.338261
[proceso 3] mi integral parcial es 0.334552
[proceso 4] mi integral parcial es 0.332716
[proceso 5] mi integral parcial es 0.336033
[0/6 tom] 2
[1/6 tom] 2
[2/6 tom] 2
[3/6 tom] 2
[4/6 tom] 2
[5/6 tom] 2
$
```

Observación. En los casos con 4 y 5 procesos, la cantidad de particiones P no es múltiplo de la cantidad de procesos N.

Al construir los objetos globales `K` y `M` o `b`, FeenoX utiliza un algoritmo similar para seleccionar qué procesos calculan las matrices elementales de cada elemento. Una vez que cada proceso ha terminado de barrer los elementos locales, se le pide a PETSc que ensamble la matriz global enviando información no local a través de mensajes MPI de forma tal de que cada proceso tenga filas contiguas de los objetos globales.

Observación. Cada fila de la matriz global `K` corresponde a un grado de libertad asociado a un nodo de la malla.

1.3.5. Ejecución en la nube

No es sólo poder hacer `mpirun` por SSH!

Hay que poner todo en una red, configurar nfs, hostfiles, et.c

Pero además tener en cuenta interacción remota en tiempo de ejecución

- cómo reportar el estado del solver a demanda
  - en un gui web
  - en un email
  - en un whatsapp

GUIs

- web

- desktop
- mobile

Client

- python to do the auth + versioning + launch + follow the execution + get results

1.3.6. Extensibildiad

- `src/pdes/`
- FVM
- truss1d

El código es GPLv3+. El + es por extensibilidad.

1.3.7. Integración continua

Github actions

TODO: code coverage?

1.3.8. Documentación

Markdown

Pandoc:

- HTML
- PDF (a través de LaTeX)
- Github Markdown (READMEs)

`man feenox`

show `INTEGRATE`

SDS

SRS

Contributing

Code of conduct

2. Resultados

> A good hockey player plays where the puck is.
> A great hockey player plays where the puck is going to be.
> *Wayne Gretzky*

> Es el “tocate una que sepamos todos” de S_N.
> *Dr. Ignacio Márquez Damián, sobre el problema de Reed (2023)*

Cada uno de estos diez problemas no puede ser resuelto con una herramienta computacional neutrónica de nivel núcleo que no soporte alguno de los cuatro puntos distintivos de FeenoX:

a. Filosofía Unix, especialmente integración en scripts
b. Mallas no estructuradas
c. Ordenadas discretas (además de difusión)
d. Paralelización en varios nodos de cálculo

Problema	Unix	Mallas	S_N	Paralelización
Mallas no conformes (2.1)	●	●		◒
Reed (2.2)	○	◒	●	
IAEA 2D PWR (2.3)	◒	●		
Azmy (2.4)	●	●	●	○
Los Alamos (2.5)	●	◒	●	
Slab a dos zonas (2.6)	●	●		
Cubo-esfera (2.7)	●	●		
Pescaditos (2.8)	●	●	○	
Stanford bunny (2.9)	●	●	○	
Vertical PHWR (2.10)		●	◒	●

- ● requerido
- ◒ recomendado
- ○ opcional

Observación. Ver apéndice para más problemas.

2.1. Mapeo en mallas no conformes

> **TL;DR**: Sobre la importancia de que FeenoX siga la filosofía Unix.

Este prime caso no resuelve ninguna PDE pero sirve para ilustrar…

1. las ideas de la filosofía Unix [5], [6], en particular programas que…
    - hagan una cosa y que la hagan bien
    - trabajen juntos
    - manejen flujos[1] de texto porque esa es una interfaz universal.
2. la capacidad de FeenoX de leer distribuciones espaciales definidas sobre los nodos de una cierta malla no estructurada y de evaluarla en posiciones **x** arbitrarias.

Una aplicación de esta segunda característica es leer una distribución espacial de temperaturas calculadas por un solver térmico (el mismo FeenoX podría servir) y utilizarlas para construir la matriz de rigidez de otro problema (por ejemplo elasticidad lineal para problemas termo-mecánicos o transporte o difusión de neutrones para neutrónica realimentada con termohidráulica). En este caso, los puntos de evaluación son los puntos de Gauss de los elementos de la segunda malla.

En este problema escribimos una función $f(x,y,z)$ definida algebraicamente en los nodos de un cubo unitario $[0,1]\times[0,1]\times[0,1]$ creado en Gmsh con la instrucción de OpenCASCADE:

```
SetFactory("OpenCASCADE");
Box(1) = {0, 0, 0, 1, 1, 1};
```

mallado con un algoritmo completamente no estructurado utilizando una cierta cantidad n_1 de elementos por lado. Luego, se lee esa malla de densidad c_1 con los valores nodales de $f(\mathbf{x})$ y los interpolamos en la posición de los nodos del mismo cubo mallado con otra densidad n_2. Como hemos partido de una función algebraica, podemos evaluar el error cometido en la interpolación en función de las densidades n_1 y n_2.

Observación. Este procedimiento no es exactamente el necesario para realizar cálculos acoplados ya que la evaluación en la segunda malla es sobre los nodos y no sobre los puntos de Gauss, pero el concepto es el mismo: interpolar valores nodales en puntos arbitrarios.

El script `run.sh` realiza una inicialización y tres pasos:

0. Lee de la línea de comandos la función $f(x,y,z)$. Si no se provee ninguna, utiliza

$$f(x,y,z) = 1 + x\cdot\sqrt{y} + 2\cdot\log(1+y+z) + \cos(xz)\cdot e^{y\cdot z}$$

1. Crea cinco mallas con $n = 10, 20, 30, 40, 50$ elementos por lado a partir del cubo base. Cada una de estas cinco mallas `cube-n.msh` (donde `n` es 10, 20, 30, 40 o 50) es leída por FeenoX y se crea un archivo nuevo llamado `cube-n-src.msh` con un campo escalar `f` definido sobre los nodos según el argumento pasado por `run.sh` a FeenoX en `$1`:

[1] Del inglés *streams*.

```
READ_MESH cube-$2.msh
f(x,y,z) = $1
WRITE_MESH cube-$2-src.msh f
```

2. Para cada combinación $n_1 = 10, \dots, 50$ y $n_2 = 10, \dots, 50$, lee la malla `cube-n1-src.msh` con el campo escalar `f` y define una función $f(x, y, z)$ definida por puntos en los nodos de la malla de entrada. Entonces escribe un archivo de salida VTK llamado `cube-n1-n2-dst.vtk` con dos campos escalares nodales:

   1. la función $f(x, y, z)$ de la malla de entrada interpolada en la malla de salida
   2. el valor absoluto de la diferencia entre la $f(x, y, z)$ interpolada y la expresión algebraica original de referencia:

```
READ_MESH cube-$2-src.msh DIM 3 READ_FUNCTION f
READ_MESH cube-$3.msh
WRITE_MESH cube-$2-$3-dst.vtk f NAME error "abs(f(x,y,z)-($1))" MESH cube-$3.msh
```

3. Finalmente, para cada archivo VTK, lee el campo escalar como `f_msh` y calcula el error L_2 como

$$e_2 = \int \sqrt{\left[f_{\text{msh}}(\mathbf{x}) - f(\mathbf{x})\right]^2}\, d^3\mathbf{x}$$

   y el error L_∞ como

$$e_\infty = \max \left| f_{\text{msh}}(\mathbf{x}) - f(\mathbf{x}) \right|$$

   e imprime una línea con un formato adecuado para que el script `run.sh` pueda escribir una tabla Markdown que pueda ser incluida en un archivo de documentación con control de versiones Git, tal como esta tesis de doctorado.

```
READ_MESH cube-$2-src.msh DIM 3 READ_FUNCTION f
READ_MESH cube-$3.msh
WRITE_MESH cube-$2-$3-dst.vtk f NAME error "abs(f(x,y,z)-($1))" MESH cube-$3.msh
```

tom:

n	elementos	nodos	tiempo de mallado [s]	tiempo de rellenado [s]
10	4.979	1.201	0,09	0,02
20	37.089	7.411	0,50	0,11
30	123.264	22.992	2,09	0,58
40	289.824	51.898	5,71	1,68
50	560.473	98.243	12,12	3,63

ansys:

n	elementos	nodos	tiempo de mallado [s]	tiempo de rellenado [s]
10	4.979	1.201	0,09	0,01
20	37.089	7.411	0,41	0,05
30	123.264	22.992	1,19	0,26
40	289.824	51.898	3,23	0,87
50	560.473	98.243	7,04	1,85

tom:

n_1	n_2	error L_2	error L_∞	tiempo [s]
10	10	$1.3 \cdot 10^{-2}$	$6.2 \cdot 10^{-6}$	0.04
10	20	$1.3 \cdot 10^{-2}$	$9.0 \cdot 10^{-2}$	0.17
10	30	$1.3 \cdot 10^{-2}$	$9.6 \cdot 10^{-2}$	0.73
10	40	$1.3 \cdot 10^{-2}$	$9.4 \cdot 10^{-2}$	2.01
10	50	$1.3 \cdot 10^{-2}$	$9.8 \cdot 10^{-2}$	4.18
20	10	$1.3 \cdot 10^{-2}$	$4.1 \cdot 10^{-3}$	0.13
20	20	$6.2 \cdot 10^{-3}$	$6.9 \cdot 10^{-6}$	0.21
20	30	$6.4 \cdot 10^{-3}$	$6.4 \cdot 10^{-2}$	0.92
20	40	$6.2 \cdot 10^{-3}$	$6.7 \cdot 10^{-2}$	2.34
20	50	$6.1 \cdot 10^{-3}$	$6.7 \cdot 10^{-2}$	4.70
30	10	$1.3 \cdot 10^{-2}$	$1.7 \cdot 10^{-3}$	0.58
30	20	$6.4 \cdot 10^{-3}$	$6.4 \cdot 10^{-3}$	0.78
30	30	$4.2 \cdot 10^{-3}$	$7.1 \cdot 10^{-6}$	1.22
30	40	$4.3 \cdot 10^{-3}$	$4.7 \cdot 10^{-2}$	3.07
30	50	$4.2 \cdot 10^{-3}$	$5.3 \cdot 10^{-2}$	5.64
40	10	$1.3 \cdot 10^{-2}$	$1.2 \cdot 10^{-3}$	1.64
40	20	$6.3 \cdot 10^{-3}$	$5.3 \cdot 10^{-3}$	1.84
40	30	$4.3 \cdot 10^{-3}$	$1.3 \cdot 10^{-2}$	2.63
40	40	$3.1 \cdot 10^{-3}$	$7.4 \cdot 10^{-6}$	3.55
40	50	$3.2 \cdot 10^{-3}$	$3.6 \cdot 10^{-2}$	7.11
50	10	$1.3 \cdot 10^{-2}$	$6.0 \cdot 10^{-4}$	3.37
50	20	$6.2 \cdot 10^{-3}$	$2.1 \cdot 10^{-3}$	3.59
50	30	$4.2 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$	4.45
50	40	$3.2 \cdot 10^{-3}$	$2.4 \cdot 10^{-2}$	6.28
50	50	$2.5 \cdot 10^{-3}$	$7.3 \cdot 10^{-6}$	7.40

ansys:

n_1	n_2	error L_2	error L_∞	tiempo [s]
10	10	1.3×10^{-2}	6.2×10^{-6}	0.02
10	20	1.3×10^{-2}	9.0×10^{-2}	0.08
10	30	1.3×10^{-2}	9.6×10^{-2}	0.32
10	40	1.3×10^{-2}	9.4×10^{-2}	1.01

n_1	n_2	error L_2	error L_∞	tiempo [s]
10	50	1.3×10^{-2}	9.8×10^{-2}	1.94
20	10	1.3×10^{-2}	4.1×10^{-3}	0.06
20	20	6.2×10^{-3}	6.9×10^{-6}	0.11
20	30	6.4×10^{-3}	6.4×10^{-2}	0.40
20	40	6.2×10^{-3}	6.7×10^{-2}	0.98
20	50	6.1×10^{-3}	6.7×10^{-2}	2.30
30	10	1.3×10^{-2}	1.7×10^{-3}	0.29
30	20	6.4×10^{-3}	6.4×10^{-3}	0.36
30	30	4.2×10^{-3}	7.1×10^{-6}	0.57
30	40	4.3×10^{-3}	4.7×10^{-2}	1.48
30	50	4.2×10^{-3}	5.3×10^{-2}	2.78
40	10	1.3×10^{-2}	1.2×10^{-3}	0.99
40	20	6.3×10^{-3}	5.3×10^{-3}	1.06
40	30	4.3×10^{-3}	1.3×10^{-2}	1.44
40	40	3.1×10^{-3}	7.4×10^{-6}	1.95
40	50	3.2×10^{-3}	3.6×10^{-2}	3.79
50	10	1.3×10^{-2}	6.0×10^{-4}	2.07
50	20	6.2×10^{-3}	2.1×10^{-3}	2.31
50	30	4.2×10^{-3}	3.9×10^{-3}	2.62
50	40	3.2×10^{-3}	2.4×10^{-2}	3.74
50	50	2.5×10^{-3}	7.3×10^{-6}	4.26

Observación. El L_∞ se hace sobre los nodos y sobre los puntos de Gauss. Recordar la figura 1.3.

Observación. Si $f(\mathbf{x})$ fuese lineal o incluso polinómica, los errores serían mucho menores.

Observación. Performance. En el repositorio https://github.com/gtheler/feenox-non-conformal-mesh-interpolation hay más data.

Observación. Performance. Comparación con Ansys.

2.2. El problema de Reed

> **TL;DR**: Este problema tiene curiosidad histórica, es uno de los problemas más sencillos no triviales que podemos encontrar y sirve para mostrar que para tener en cuenta regiones vacías no se puede utilizar una formulación de difusión.

2.3. IAEA PWR Benchmark

> **TL;DR**: El problema original de 1976 propone resolver un cuarto de núcleo cuando en realidad la simetría es 1/8.

2.3.1. Caso 2D original

2.3.2. Caso 2D con simetría 1/8

2.3.3. Caso 2D con reflector circular

2.3.4. Caso 3D original con simetría 1/8

2.4. El problema de Azmy

> **TL;DR**: Este problema ilustra el "efecto rayo" de la formulación de ordenadas discretas en dos dimensiones. Para estudiar completamente el efecto se necesita o rotar la geometría con respecto a las direcciones de S_N.

2.5. Benchmarks de criticidad de Los Alamos

> **TL;DR**: Curiosidad histórica y verificación con el método de soluciones exactas.

2.6. Slab a dos zonas, efecto de dilución de XSs

> **TL;DR**: Este problema ilustra el error cometido al analizar casos multi-material con mallas estructuradas donde la interfaz no coincide con los nodos de la malla.

2.7. Estudios paramétricos: el reactor cubo-esfera

> **TL;DR**: No es posible resolver una geometría con bordes curvos con una malla cartesiana estructurada.

2.8. Optimización: el problema de los pescaditos

> **TL;DR**: Composición con una herramienta de optimización.

2.9. Verificación con el método de soluciones fabricadas

> **TL;DR**: Para verificar los métodos numéricos con el método de soluciones fabricadas se necesita un solver que permita definir propiedades materiales en función del espacio a través de expresiones algebraicas.

reactor tipo conejo MMS

ver thermal-slab-transient-mms-capacity-of-T.fee thermal-slab-transient-mms.fee

2.10. PHWR de siete canales y tres barras de control inclinadas

> **TL;DR**: Mallas no estructuradas, dependencias espaciales no triviales, escalabilidad.

mostrar que KSP es mucho más barato que EPS

A. Software Requirements Specification for an Engineering Computational Tool

An imaginary (a thought experiment if you will) "Request for Quotation" issued by a fictitious agency asking for vendors to offer a free and open source cloud-based computational tool to solve engineering problems. This (imaginary but plausible) Software Requirements Specification document describes the mandatory features this tool ought to have and lists some features which would be nice the tool had, following current state-of-the-art methods and technologies.

A.1. Introduction

A computational tool (herein after referred to as *the tool*) specifically designed to be executed in arbitrarily-scalable remote server (i.e. in the cloud) is required in order to solve engineering problems following the current state-of-the-art methods and technologies impacting the high-performance computing world. This (imaginary but plausible) Software Requirements Specification document describes the mandatory features this tool ought to have and lists some features which would be nice the tool had. Also it contains requirements and guidelines about architecture, execution and interfaces in order to fulfill the needs of cognizant engineers as of 2022 (and the years to come) are defined.

On the one hand, the tool should allow to solve industrial problems under stringent efficiency (sección A.2.3) and quality (sección A.4) requirements. It is therefore mandatory to be able to assess the source code for

- independent verification, and/or
- performance profiling, and/or
- quality control

by qualified third parties from all around the world, so it has to be *open source* according to the definition of the Open Source Initiative.

On the other hand, the initial version of the tool is expected to provide a basic functionality which might be extended (sección A.1.1 and sección A.2.6) by academic researchers and/or professional programmers. It thus should also be *free*—in the sense of freedom, not in the sense of price—as defined by the Free Software Foundation. There is no requirement on the pricing scheme, which is up to the vendor to define in the offer along with the detailed licensing terms. These should allow users to solve their problems the way they need and, eventually, to modify and improve the tool to suit their needs. If they cannot program themselves, they should have the *freedom* to hire somebody to do it for them.

A.1.1. Objective

The main objective of the tool is to be able to solve engineering problems which are usually casted as differential-algebraic equations (DAEs) or partial differential equations (PDEs), such as

- heat conduction
- mechanical elasticity
- structural modal analysis
- frequency studies
- electromagnetism
- chemical diffusion
- process control dynamics
- computational fluid dynamics
- …

on one or more mainstream cloud servers, i.e. computers with hardware and operating systems (futher discussed in sección A.2) that allows them to be available online and accessed remotely either interactively or automatically by other computers as well. Other architectures such as high-end desktop personal computers or even low-end laptops might be supported but they should not the main target (i.e. the tool has to be cloud-first but laptop-friendly).

The initial version of the tool must be able to handle a subset of the above list of problem types. Afterward, the set of supported problem types, models, equations and features of the tool should grow to include other models as well, as required in sección A.2.6.

A.1.2. Scope

The tool should allow users to define the problem to be solved programmatically. That is to say, the problem should be completely defined using one or more files either…

a. specifically formatted for the tool to read such as JSON or a particular input format (historically called input decks in punched-card days), and/or
b. written in an high-level interpreted language such as Python or Julia.

It should be noted that a graphical user interface is *not* required. The tool may include one, but it should be able to run without needing any interactive user intervention rather than the preparation of a set of input files. Nevertheless, the tool might *allow* a GUI to be used. For example, for a basic usage involving simple cases, a user interface engine should be able to create these problem-definition files in order to give access to less advanced users to the tool using a desktop, mobile and/or web-based interface in order to run the actual tool without needing to manually prepare the actual input files.

However, for general usage, users should be able to completely define the problem (or set of problems, i.e. a parametric study) they want to solve in one or more input files and to obtain one or more output files containing the desired results, either a set of scalar outputs (such as maximum stresses or mean temperatures), and/or a detailed time and/or spatial distribution. If needed, a discretization of the domain may to be taken as a known input, i.e. the tool is not required to create the mesh as long as a suitable mesher can be employed using a similar workflow as the one specified in this SRS.

The tool should define and document (sección A.4.6) the way the input files for a solving particular problem are to be prepared (sección A.3.1) and how the results are to be written (sección A.3.2). Any GUI, pre-processor, post-processor or other related graphical tool used to provide a graphical interface for the user should integrate in the workflow described in the preceding paragraph: a pre-processor should create the input files needed for the tool and a post-processor should read the output files created by the tool.

A.2. Architecture

The tool must be aimed at being executed unattended on remote servers which are expected to have a mainstream (as of the 2020s) architecture regarding operating system (GNU/Linux variants and other UNIX-like OSes) and hardware stack, such as

- a few Intel-compatible CPUs per host
- a few levels of memory caches
- a few gigabytes of random-access memory
- several gigabytes of solid-statee storage

It should successfully run on

- bare-metal
- virtual servers
- containerized images

using standard compilers, dependencies and libraries already available in the repositories of most current operating systems distributions.

Preference should be given to open source compilers, dependencies and libraries. Small problems might be executed in a single host but large problems ought to be split through several server instances depending on the processing and memory requirements. The computational implementation should adhere to open and well-established parallelization standards.

Ability to run on local desktop personal computers and/laptops is not required but suggested as a mean of giving the opportunity to users to test and debug small coarse computational models before launching the large computation on a HPC cluster or on a set of scalable cloud instances. Support for non-GNU/Linux operating systems is not required but also suggested.

Mobile platforms such as tablets and phones are not suitable to run engineering simulations due to their lack of proper electronic cooling mechanisms. They are suggested to be used to control one (or more) instances of the tool running on the cloud, and even to pre and post process results through mobile and/or web interfaces.

A.2.1. Deployment

The tool should be easily deployed to production servers. Both

a. an automated method for compiling the sources from scratch aiming at obtaining optimized binaries for a particular host architecture should be provided using a well-established procedures, and
b. one (or more) generic binary version aiming at common server architectures should be provided.

Either option should be available to be downloaded from suitable online sources, either by real people and/or automated deployment scripts.

A.2.2. Execution

It is mandatory to be able to execute the tool remotely, either with a direct action from the user or from a high-level workflow which could be triggered by a human or by an automated script. The calling party should be able to monitor the status during run time and get the returned error level after finishing the execution.

The tool shall provide a mean to perform parametric computations by varying one or more problem parameters in a certain prescribed way such that it can be used as an inner solver for an outer-loop optimization tool. In this regard, it is desirable if the tool could compute scalar values such that the figure of merit being optimized (maximum temperature, total weight, total heat flux, minimum natural frequency, maximum displacement, maximum von Mises stress, etc.) is already available without needing further post-processing.

A.2.3. Efficiency

It is mandatory to be able to execute the tool automatically in a remote server. The computational resources needed from this server, i.e. costs measured in

- CPU/GPU time
- random-access memory
- long-term storage
- etc.

needed to solve a problem should be comparable to other similar state-of-the-art cloud-based script-friendly finite-element tools.

A.2.4. Scalability

The tool ought to be able to start solving small problems first to check the inputs and outputs behave as expected and then allow increasing the problem size up in order to achieve to the desired accuracy of the results. As mentioned in sección A.2, large problem should be split among different computers to be able to solve them using a finite amount of per-host computational power (RAM and CPU).

A.2.5. Flexibility

The tool should be able to handle engineering problems involving different materials with potential spatial and time-dependent properties, such as temperature-dependent thermal expansion coefficients and/or non-constant densities. Boundary conditions must be allowed to depend on both space and time as well, like non-uniform pressure loads and/or transient heat fluxes.

A.2.6. Extensibility

It should be possible to add other problem types casted as PDEs (such as the Schröedinger equation) to the tool using a reasonable amount of time by one or more skilled programmers. The tool should also allow new models (such as non-linear stress-strain constitutive relationships) to be added as well.

A.2.7. Interoperability

A mean of exchanging data with other computational tools complying to requirements similar to the ones outlined in this document. This includes pre and post-processors but also other computational programs so that coupled calculations can be eventually performed by efficiently exchanging information between calculation codes.

A.3. Interfaces

The tool should be able to allow remote execution without any user intervention after the tool is launched. To achieve this goal it is required that the problem should be completely defined in one or more input files and the output should be complete and useful after the tool finishes its execution, as already required. The tool should be able to report the status of the execution (i.e. progress, errors, etc.) and to make this information available to the user or process that launched the execution, possibly from a remote location.

A.3.1. Problem input

The problem should be completely defined by one or more input files. These input files might be

a. particularly formatted files to be read by the tool in an *ad-hoc* way, and/or
b. source files for interpreted languages which can call the tool through and API or equivalent method, and/or
c. any other method that can fulfill the requirements described so far.

Preferably, these input files should be plain ASCII files in order to allow to manage changes using distributed version control systems such as Git. If the tool provides an API for an interpreted language such as Python, then the Python source used to solve a particular problem should be Git-friendly. It is recommended not to track revisions of mesh data files but of the source input files, i.e. to track the mesher's input and not the mesher's output. Therefore, it is recommended not to mix the problem definition with the problem mesh data.

It is not mandatory to include a GUI in the main distribution, but the input/output scheme should be such that graphical pre and post-processing tools can create the input files and read the output files so as to allow third parties to develop interfaces. It is recommended to design the workflow as to make it possible for the interfaces to be accessible from mobile devices and web browsers.

It is expected that 80% of the problems need 20% of the functionality. It is acceptable if only basic usage can be achieved through the usage of graphical interfaces to ease basic usage at first. Complex problems involving non-trivial material properties and boundary conditions not be treated by a GUI and only available by needing access to the input files.

A.3.2. Results output

The output ought to contain useful results and should not be cluttered up with non-mandatory information such as ASCII art, notices, explanations or copyright notices. Since the time of cognizant engineers is far more expensive than CPU time, output should be easily interpreted by either a human or, even better, by other programs or interfaces—especially those based in mobile and/or web platforms. Open-source formats and standards should be preferred over privative and ad-hoc formatting to encourage the possibility of using different workflows and/or interfaces.

A.4. Quality assurance

Since the results obtained with the tool might be used in verifying existing equipment or in designing new mechanical parts in sensitive industries, a certain level of software quality assurance is needed. Not only are best-practices for developing generic software such as

- employment of a version control system,
- automated testing suites,
- user-reported bug tracking support.
- etc.

required, but also since the tool falls in the category of engineering computational software, verification and validation procedures are also mandatory, as discussed below. Design should be such that governance of engineering data including problem definition, results and documentation can be efficiently performed using state-of-the-art methodologies, such as distributed control version systems

A.4.1. Reproducibility and traceability

The full source code and the documentation of the tool ought to be maintained under a control version system. Whether access to the repository is public or not is up to the vendor, as long as the copying conditions are compatible with the definitions of both free and open source software from the FSF and the OSI, respectively as required in sección A.1.

In order to be able to track results obtained with different version of the tools, there should be a clear release procedure. There should be periodical releases of stable versions that are required

- not to raise any warnings when compiled using modern versions of common compilers (e.g. GNU, Clang, Intel, etc.)
- not to raise any errors when assessed with dynamic memory analysis tools (e.g. Valgrind) for a wide variety of test cases
- to pass all the automated test suites as specified in sección A.4.2

These stable releases should follow a common versioning scheme, and either the tarballs with the sources and/or the version control system commits should be digitally signed by a cognizant responsible. Other unstable versions with partial and/or limited features might be released either in the form of tarballs or made available in a code repository. The requirement is that unstable tarballs and main (a.k.a. trunk) branches on the repositories have to be compilable. Any feature that does not work as expected or that does not even compile has to be committed into develop branches before being merge into trunk.

If the tool has an executable binary, it should be able to report which version of the code the executable corresponds to. If there is a library callable through an API, there should be a call which returns the version of the code the library corresponds to.

It is recommended not to mix mesh data like nodes and element definition with problem data like material properties and boundary conditions so as to ease governance and tracking of computational models and the results associated with them. All the information needed to solve a particular problem (i.e. meshes, boundary conditions, spatially-distributed material properties, etc.) should be generated from a very simple set of files which ought to be susceptible of being tracked with current state-of-the-art version control systems. In order to comply with this suggestion, ASCII formats should be favored when possible.

A.4.2. Automated testing

A mean to automatically test the code works as expected is mandatory. A set of problems with known solutions should be solved with the tool after each modification of the code to make sure these changes still give the right answers for the right questions and no regressions are introduced. Unit software testing practices like continuous integration and test coverage are recommended but not mandatory.

The tests contained in the test suite should be

- varied,
- diverse, and

- independent

Due to efficiency issues, there can be different sets of tests (e.g. unit and integration tests, quick and thorough tests, etc.) Development versions stored in non-main branches can have temporarily-failing tests, but stable versions have to pass all the test suites.

A.4.3. Bug reporting and tracking

A system to allow developers and users to report bugs and errors and to suggest improvements should be provided. If applicable, bug reports should be tracked, addressed and documented. User-provided suggestions might go into the back log or TO-DO list if appropriate.

Here, "bug and errors" mean failure to

- compile on supported architectures,
- run (unxepected run-time errors, segmentation faults, etc.)
- return a correct result

A.4.4. Verification

Verification, defined as

> The process of determining that a model implementation accurately represents the developer's conceptual description of the model and the solution to the model.

i.e. checking if the tool is solving right the equations, should be performed before applying the code to solve any industrial problem. Depending on the nature and regulation of the industry, the verification guidelines and requirements may vary. Since it is expected that code verification tasks could be performed by arbitrary individuals or organizations not necessarily affiliated with the tool vendor, the source code should be available to independent third parties. In this regard, changes in the source code should be controllable, traceable and well documented.

Even though the verification requirements may vary among problem types, industries and particular applications, a common method to verify the code is to compare solutions obtained with the tool with known exact solutions or benchmarks. It is thus mandatory to be able to compare the results with analytical solutions, either internally in the tool or through external libraries or tools. This approach is called the Method of Exact Solutions and it is the most widespread scheme for verifying computational software, although it does not provide a comprehensive method to verify the whole spectrum of features. In any case, the tool's output should be susceptible of being post-processed and analysed in such a way to be able to determine the order of convergence of the numerical solution as compared to the exact one.

Another possibility is to follow the Method of Manufactured Solutions, which does address all the shortcomings of MES. It is highly encouraged that the tool allows the application of MMS for software verification. Indeed, this method needs a full explanation of the equations solved by the tool, up to the point that [21] says that

> Difficulties in determination of the governing equations arises frequently with commercial software, where some information is regarded as proprietary. If the governing equations cannot be determined, we would question the validity of using the code.

To enforce the availability of the governing equations, the tool has to be open source as required in sección A.1 and well documented as required in sección A.4.6.

A report following either the MES and/or MMS procedures has to be prepared for each type of equation that the tool can solve. The report should show how the numerical results converge to the exact or manufactured results with respect to the mesh size or number of degrees of freedom. This rate should then be compared to the theoretical expected order.

Whenever a verification task is performed and documented, at least one of the cases should be added to the test suite. Even though the verification report must contain a parametric mesh study, a single-mesh case is enough to be added to the test suite. The objective of the tests defined in sección A.4.2 is to be able to detect regressions which might have been inadvertently introduced in the code and not to do any actual verification. Therefore a single-mesh case is enough for the test suites.

A.4.5. Validation

As with verification, for each industrial application of the tool there should be a documented procedure to perform a set of validation tests, defined as

> The process of determining the degree to which a model is an accurate representation of the real world from the perspective of the intended uses of the model.

i.e. checking that the right equations are being solved by the tool. This procedure should be based on existing industry standards regarding verification and validation such as ASME, AIAA, IAEA, etc. There should be a procedure for each type of physical problem (thermal, mechanical, thermo-mechanical, nuclear, etc.) and for each problem type when a new

- geometry,
- mesh type,
- material model,
- boundary condition,
- data interpolation scheme

or any other particular application-dependent feature is needed.

A report following the validation procedure defined above should be prepared and signed by a responsible engineer in a case-by-case basis for each particular field of application of the tool. Verification can be performed against

- known analytical results, and/or
- other already-validated tools following the same standards, and/or
- experimental results.

A.4.6. Documentation

Documentation should be complete and cover both the user and the developer point of view. It should include a user manual adequate for both reference and tutorial purposes. Other forms of simplified documentation such as quick reference cards or video tutorials are not mandatory but highly recommended. Since the tool should be extendable (sección A.2.6), there should be a separate development manual covering the programming design and implementation, explaining how to extend the code and how to add new features. Also, as non-trivial mathematics which should be verified (sección A.4.4) are expected, a thorough explanation of what equations are taken into account and how they are solved is required.

It should be possible to make the full documentation available online in a way that it can be both printed in hard copy and accessed easily from a mobile device. Users modifying the tool to suit their own needs should be able to modify the associated documentation as well, so a clear notice about the licensing terms of the documentation itself (which might be different from the licensing terms of the source code itself) is mandatory. Tracking changes in the documentation should be similar to tracking changes in the code base. Each individual document ought to explicitly state to which version of the tool applies. Plain ASCII formats should be preferred. It is forbidden to submit documentation in a non-free format.

The documentation shall also include procedures for

- reporting errors and bugs
- releasing stable versions
- performing verification and validation studies
- contributing to the code base, including
  - code of conduct
  - coding styles
  - variable and function naming convections

Referencias

> Cuando se proclamó que la Biblioteca abarcaba todos los libros, la primera impresión fue de extravagante felicidad. Todos los hombres se sintieron señores de un tesoro intacto y secreto.
> *Jorge Luis Borges, La Biblioteca de Babel, 1941*

[1] S. Williams, *Free as in Freedom: Richard Stallman's Crusade for Free Software*. O'Reilly, 2002.

[2] E. S. Raymond, *The Cathedral and the Bazaar*, Second. O'Reilly, 2001.

[3] R. Stallman, «The GNU Manifesto», *j-ddj*, vol. 10, n.º 3, pp. 30-??, mar. 1985.

[4] T. Vadén y R. M. Stallman, *The Hacker Community and Ethics: An Interview with Richard M. Stallman*. Tampere University Press, 2002.

[5] D. Ritchie y K. Thompson, «The UNIX time-sharing system», *Communications of the ACM*, vol. 7, n.º 17, pp. 365-375, 1974.

[6] E. S. Raymond, *The Art of UNIX Programming*. Addison-Wesley, 2003.

[7] G. Theler, «On the design basis of a new core-level neutronic code written from scratch», *Mecánica Computacional*, vol. XXXIII, n.º Number 48, Numerical Methods in Reactor Physics (B), pp. 3169-3194, 2014.

[8] S. Balay *et al.*, «PETSc/TAO Users Manual», Argonne National Laboratory, ANL-21/39 - Revision 3.19, 2023.

[9] S. Balay, W. D. Gropp, L. C. McInnes, y B. F. Smith, «Efficient Management of Parallelism in Object Oriented Numerical Software Libraries», en *Modern Software Tools in Scientific Computing*, 1997, pp. 163-202.

[10] J. E. Roman, C. Campos, L. Dalcin, E. Romero, y A. Tomas, «SLEPc Users Manual», D. Sistemes Informàtics i Computació, Universitat Politècnica de València, DSIC-II/24/02 - Revision 3.19, 2023.

[11] V. Hernandez, J. E. Roman, y V. Vidal, «SLEPc: A Scalable and Flexible Toolkit for the Solution of Eigenvalue Problems», *ACM Trans. Math. Software*, vol. 31, n.º 3, pp. 351-362, 2005.

[12] G. Theler, F. J. Bonetto, y A. Clausse, «Optimización de parametros en reactores de potencia: base de diseño del codigo neutrónico milonga», *Reunion Anual de la Asociacion Argentina de Tecnologia Nuclear*, vol. XXXVII, 2010.

[13] G. Theler, F. J. Bonetto, y A. Clausse, «Solution of the 2D IAEA PWR Benchmark with the neutronic code milonga», *Actas de la Reunión Anual de la Asociación Argentina de Tecnología Nuclear*, vol. XXXVIII, 2011.

[14] D. E. Knuth, *Selected Papers on Computer Languages*. Center for the Study of Language; Information, 2003.

[15] M. Steffen, «A simple method for monotonic interpolation in one dimension», *Astron. Astrophys.*, n.º 239, pp. 443-450, 1990.

[16] A. H. Baker, Tz. V. Kolev, y U. M. Yang, «Improving algebraic multigrid interpolation operators for linear elasticity problems», *Numerical Linear Algebra With Applications*, vol. 17, pp. 495-517, 2010.

[17] C. A. Felippa, *Introduction to Finite Element Methods*. University of Colorado Boulder, 2004.

[18] G. Theler, «Controladores basados en lógica difusa y loops de convección natural caóticos». Proyecto Integrador de la Carrera de Ingeniería Nuclear, Instituto Balseiro, 2007.

[19] G. Theler, «Análisis no lineal de inestabilidades en el problema acoplado termohidráulico-neutrónico». Tesis de la Carrera de Maestría en Ingeniería, Instituto Balseiro, 2008.

[20] D. Shepard, «A two-dimensional interpolation function for irregularly-spaced data», en *Proceedings of the 1968 ACM National Conference*, 1968, pp. 517-524.

[21] K. Salari y P. Knupp, «Code Verification by the Method of Manufactured Solutions», Sandia Natioal Laboratories, SAMD2000-1444, 2000.