第50回炉物理夏期セミナー「炉物理プログラミングの「今」を学ぶ」

ROBUTSURI PROGRAMMING DOJO

M.TATSUMI@NEL

講師のプログラミング遍歴

- ‘80年代：マイコン・パソコンで遊んでいた【BASIC, マシン語, C】
- ‘90年代前半：大学～大学院時代【Fortran, C, C++, Lisp, Perl他】
- ‘90年代後半：NFI時代にアジャイル開発に出会う【C++, Perl】
- ‘00年代：コード開発プロジェクト多数【C++, Python, PHP, SQL, Ruby】
- ‘10年代：データ管理/機械学習【Python, Non-SQL, Javascript】

一番長く使っている言語：C++
一番好きな言語：Ruby
今一番使っている言語：Python
実は最強だと思う言語：Javascript
勉強しておくべき言語：　？？？

おすすめ記事

2018年の人気プログラミング言語の徹底比較

https://techacademy.jp/magazine/8735

DOJOの狙い

炉物理の問題解決を通じて、オブジェクト指向プログラミングの実践的アプローチを伝授する

- 炉物理
  - 応答行列法による固有値問題
  - 1群計算と2群計算の違い
  - 裸の体系と反射体付き体系の違い
- プログラミング
  - オブジェクト指向プログラミング
  - テスト駆動開発
  - Python によるコーディング

DOJOの流れ

1. 理論編
   - 事前に読んできていることを前提とする
   - 本当に重要な部分のみを説明する

2. 実践編
   - プログラム設計における考え方の説明
   - 1群計算コードのテスト駆動開発を「3分間クッキング方式」で体験する
   - 2群計算への拡張と加速法の導入を体験する

特に重要な概念

テキストで事前学習しているはずなので説明は省略

- デザインパターンとアンチパターン
- アジャイル開発
  - テスト駆動型開発 (Test-Driven Development)
  - リポジトリ管理
  - リファクタリング
- KISS, YAGNI, DRY, CoC
- 名前重要 →「プログラムは名前が9割」
- プログラムのライフサイクル
- 検証と妥当性確認
- レバレッジ戦略

いま時間がある人は・・・これを読むべし

- 一次元体系における中性子拡散固有値方程式の解析解
  - エネルギー1群・1領域・平板体系
    - テキスト P.107 ～　P.108
  - エネルギー1群・2領域・平板体系
    - テキスト P.108　～ P.109
- 応答行列法を用いた中性子拡散方程式の解法
  - P.120 の上の囲み部分 (式(19)～式(22))
  - 応答行列法を用いた中性子拡散方程式の解法
    - P.120～P.122

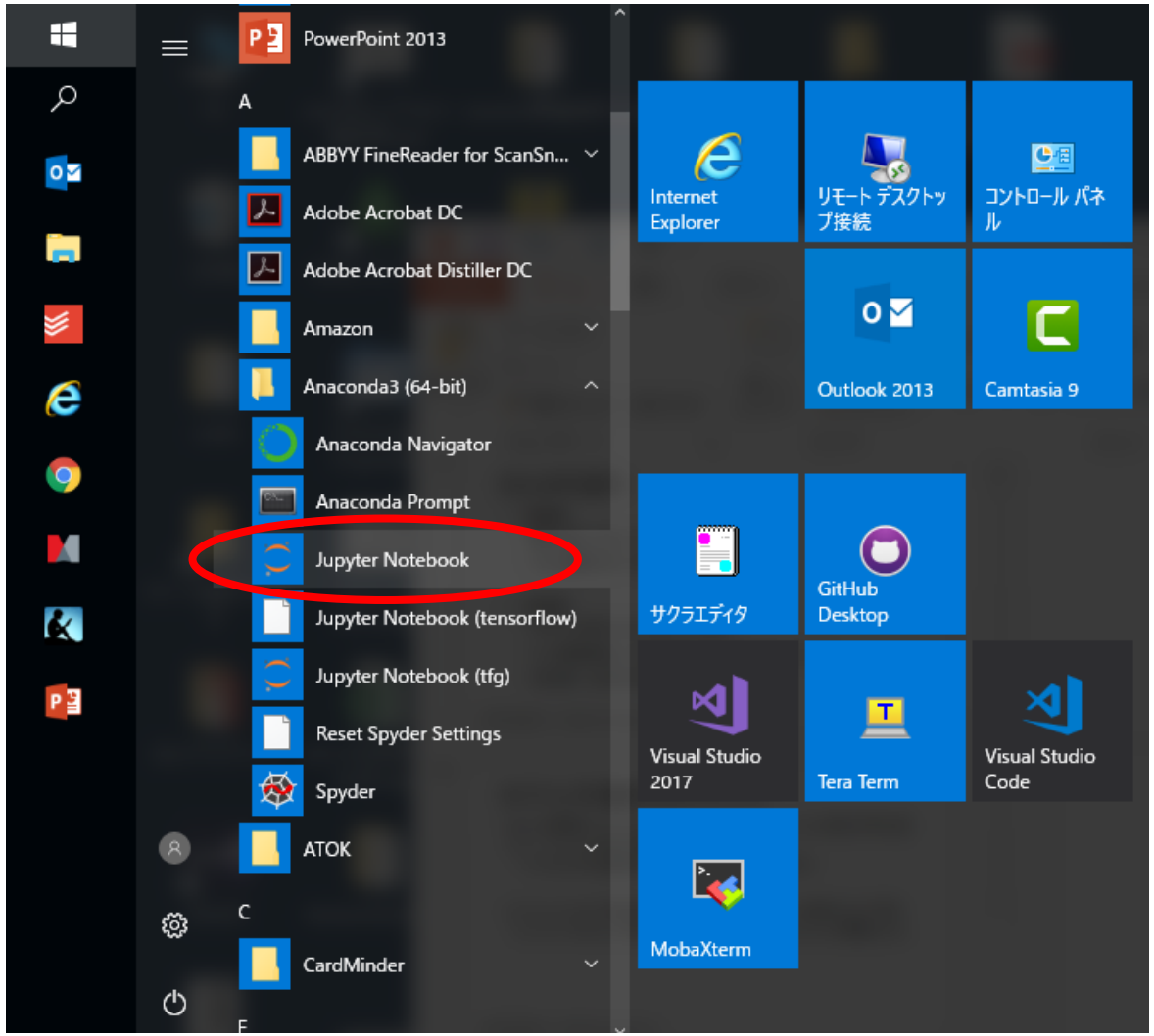
オブジェクト指向プログラミング

- 全ての要素は「オブジェクト」であり、各オブジェクトの相互作用を通じて、システム全体の挙動を制御する
  - オブジェクトの各個体を「インスタンス (instance)」と呼ぶ

- オブジェクトは「内部状態 (internal state)」とそれを操作する「手段（メソッド, method)」を持っており、「クラス (class)」として定義される。

Python 言語のおさらい

- Jupyter Notebookを起動
  - Anaconda3
    - Jupyter Notebook



Python 言語のおさらい

- 変数・演算
- リスト・辞書・タプル
- 判断 (if文)
- 繰り返し (for, while)
- 関数定義
- モジュール
- クラス定義

1群1次元拡散ソルバーの設計と実装

- オブジェクト指向分析を行う場合、現実世界をプログラムの世界に（ある程度）直接的にマッピングする方法は有効
- 良い設計は、直感的に理解出来る。
- 「将来の自分」＝「他人」という観点が重要

【設計中】

取り扱うべき対象は計算体系だから、マクロからミクロに視点をずらしていけば、その本質は、媒質の中で断面積と反応しながら中性子が拡散する･･･ということだな。

だったら、中性子や断面積をオブジェクトとして表現すれば良いかな？
ほんでもって、これ以上に分割できそうにないから、これが最も詳細なレベルのサブシステムということで良いかな？

よし、じゃぁ仮に Neutron クラス、CrossSection クラスとすることにしよう。

クラス

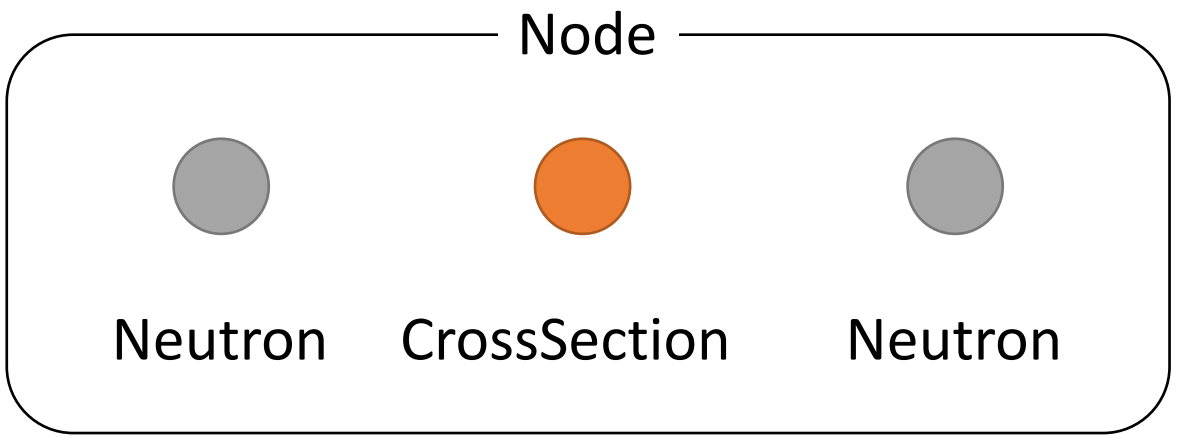
Neutron　　CrossSection

【設計中】

そしたら、この中性子や断面積といったオブジェクトは、誰がどのようにして取り扱えば良いだろう･･･。今度はミクロ（詳細）なレベルから、マクロな（より上位の）レベルに視点を戻してみよう。

計算機で取り扱うには、特に今回のような決定論的手法だったら、どうしても避けられないのが離散化だよね～。ということは、媒質をメッシュとかノードに分割する必要があるよね。であれば、仮に計算ノードという単位を作って、この中に中性子や断面積が含まれるとすれば、サブシステムとして都合が良いんじゃないかな？きっとそうだ。じゃぁ、仮に Node クラスとしておこう。

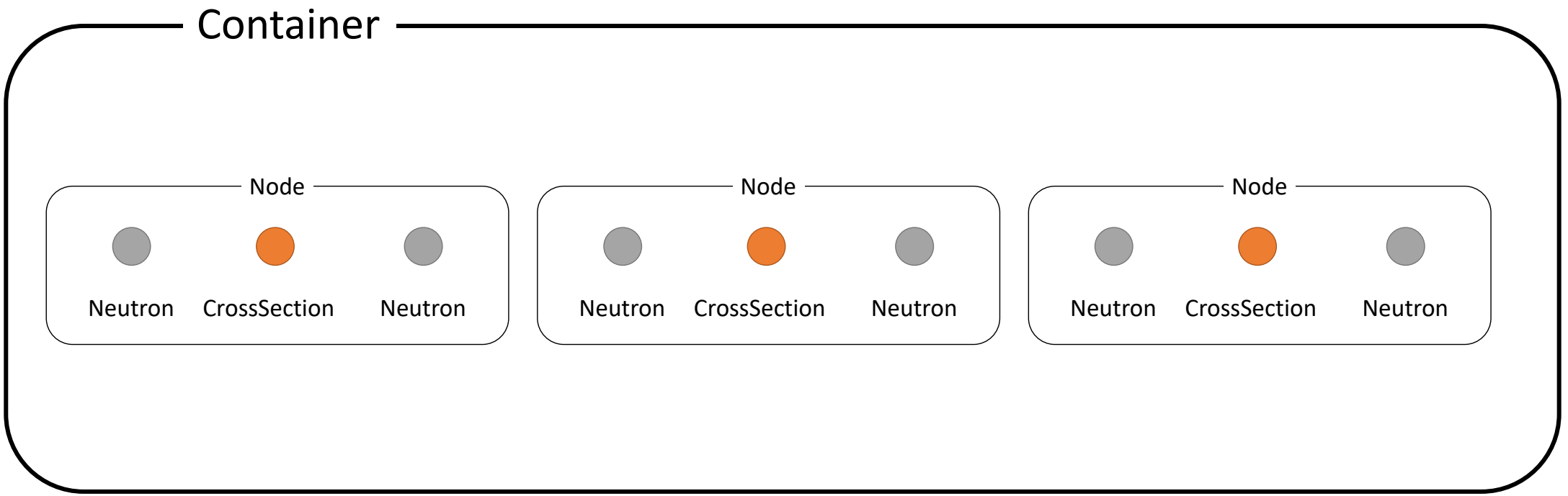
クラス



【設計中】

となると、計算体系というのは計算メッシュの集まり考えたら都合が良いかもしれないな・・・。だったら、Container クラスとして、その内部に Node オブジェクトを管理しておけば良いんじゃないだろうか！？

クラス


Container
Node
Neutron
CrossSection
Neutron
Node
Neutron
CrossSection
Neutron
Node
Neutron
CrossSection
Neutron

【設計中】

Container オブジェクトは計算体系を表すから、もしかしたら、詳細メッシュ分割した Container と、粗いメッシュ分割をした Container を別々に準備して、それぞれを上手く作用させたら粗メッシュ拡散加速法とかもうまく実現できるかもしれないな！　どうすれば良いかよく分からんけど、感覚的にはそんな感じかな！？

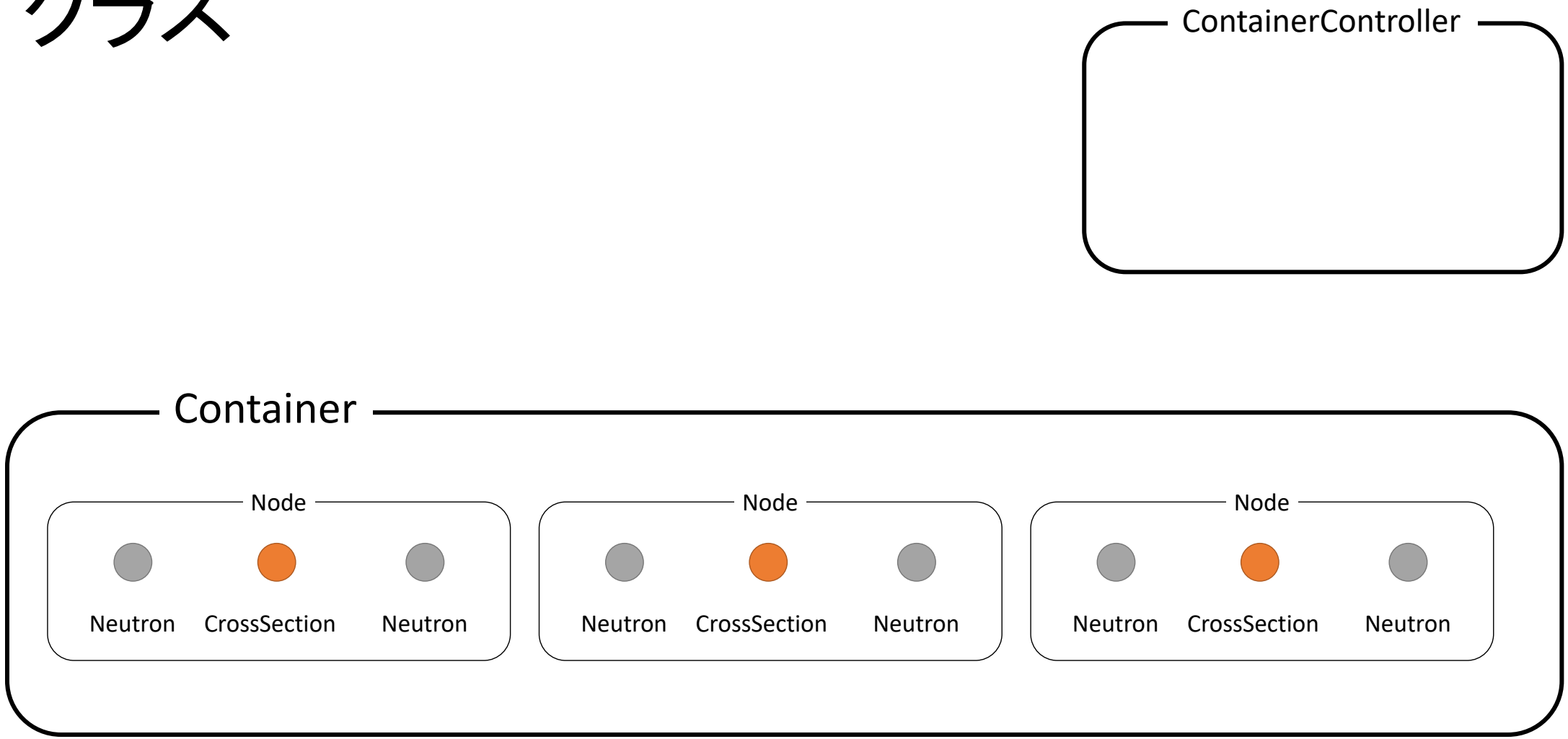
【設計中】

でも、ちょっと待てよ。Container オブジェクトに計算制御部分も入れるとなると、かなり複雑なシステムになってしまうなぁ。だったら MVC モデルのように、物理量を保持するモデルクラスと実際の挙動を取り仕切る制御クラスと分離してあげた方がよいかもしれないなぁ。

Container クラスを制御するクラスだから ContainerController だな！

我ながら安直な命名だけど、分かりやすくて良いよね～。Container オブジェクトのインスタンス1つにつき、ContainerController オブジェクトのインスタンスを用意すれば良いね。

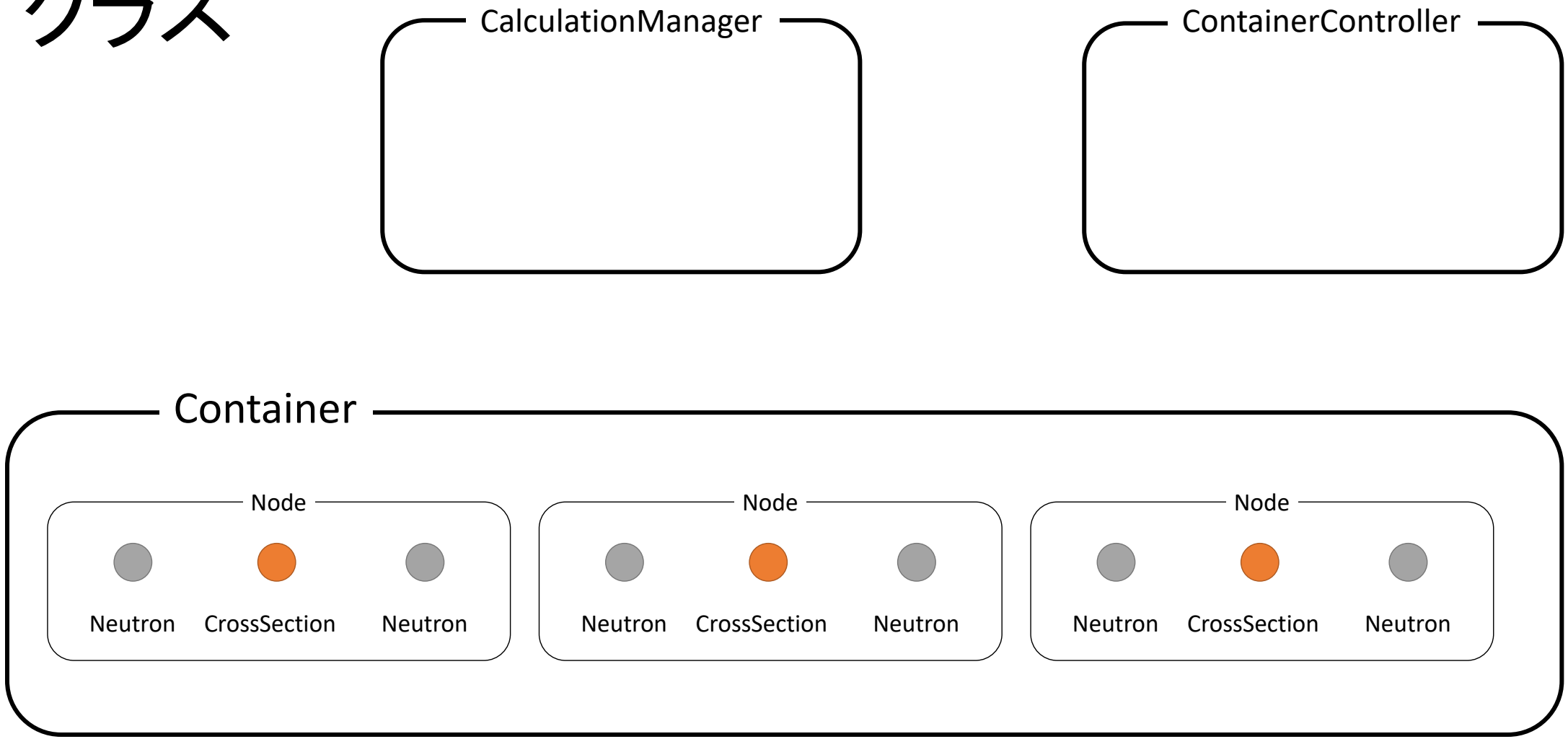
クラス


ContainerController
Container
Node
Neutron
CrossSection
Neutron
Node
Neutron
CrossSection
Neutron
Node
Neutron
CrossSection
Neutron

【設計中】

じゃぁ、計算全体をマネジメントするのはどうすれば良いかな～。安直だけど、CalclationManager で良いだろう。うん、Simple is best!

クラス


CalculationManager
ContainerController
Container
Node
Neutron
CrossSection
Neutron
Node
Neutron
CrossSection
Neutron
Node
Neutron
CrossSection
Neutron

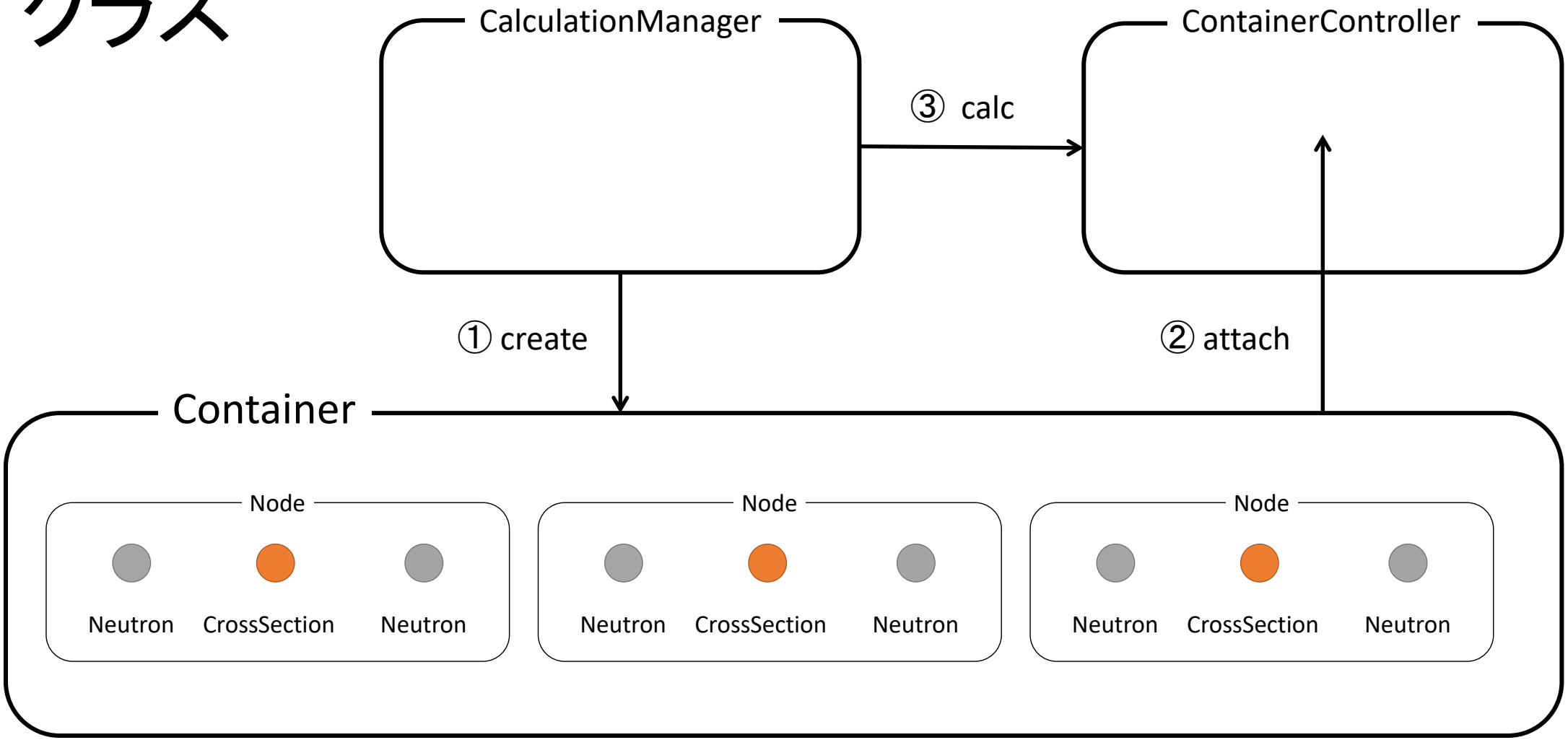
【設計中】

じゃぁ今度は全体の流れを考えてみるか。

CalculationManager が Container オブジェクトを作って、
ContainerController オブジェクトに渡してあげてから、ContainerController オブジェクトに制御を頼めば良いよね・・・。

Containerオブジェクトが生まれたときに、内部に Node オブジェクトを準備する必要があるな。これには CrossSection オブジェクトが必要かな？
これは計算条件として与えられるとして、これを Node オブジェクトに渡せばいいね。

クラス


CalculationManager
ContainerController
③ calc
① create
② attach
Container
Node
Neutron
CrossSection
Neutron
Node
Neutron
CrossSection
Neutron
Node
Neutron
CrossSection
Neutron

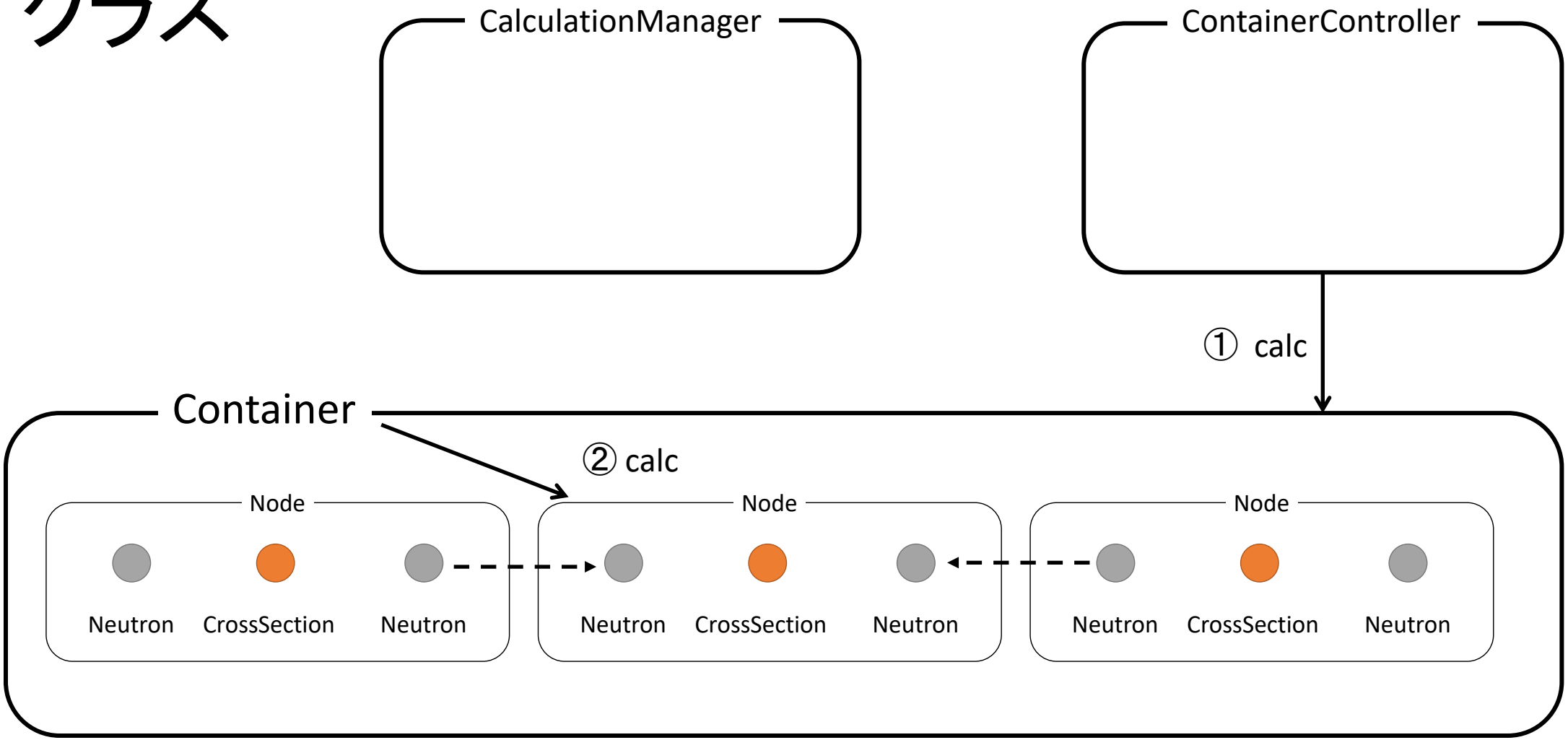
【設計中】

Container は ContainerController から「計算しろ」と言われたら、Container オブジェクトは内部の Node オブジェクト間で Neutron オブジェクトをやりとりしてから、Node オブジェクトに「計算しろ」と言えばよさそうだな。

Node オブジェクトは隣のノードから受け取った流入中性子と、断面積から計算された応答行列を使えば、放出中性子を計算できそうだな・・・。

うん、応答行列法を使えば、上手く表現できそうだ。

クラス


CalculationManager
ContainerController
① calc
Container
② calc
Node
Neutron
CrossSection
Neutron
Node
Neutron
CrossSection
Neutron
Node
Neutron
CrossSection
Neutron

【設計中】

いや、ちょっとまてよ。

中性子のために Neutron クラスってのを考えたけど、これって必要かな？　拡散理論の応答行列法なら、中性子流も中性子束も P0成分だけだから、単なるスカラー量として表現できるよなぁ。

それに、中性子オブジェクトとしたところで、特に振る舞いを規定することもないから、単なる配列とかで良いんじゃないか？

クラス構成

断面積 (CrossSection)

- 断面積データの設定・保持
- 他の断面積との足し算・引き算
- 中性子束との掛け算
- スカラー量（体積等）との掛け算
- 断面積バランスのチェック

計算体系 (Container)

- ノードの保持
- 内側反復の計算
- 体系内の総核分裂源の計算

計算ノード (Node)

- 各種物理量の保持
  - 断面積
  - 平均中性子束
- 各面の部分中性子流(流入及び放出）
- 部分中性子流のレスポンス計算
- ノード内の核分裂源の計算

計算体系制御 (ContainerController)

- 計算体系の保持
- 内側反復計算の制御

計算全体管理 (CalculationManager)

- 計算条件の管理
- 計算体系制御の管理
- 外側反復計算の制御
- 加速計算の制御

CrossSection クラスの実装

Node クラスの実装

GIT リポジトリの準備

事前に準備している人はスキップ

- 作業ディレクトリの作成

```
mkdir work
cd work
```

- robutsuri リポジトリのクローン

```
git clone  https://github.com/tatsumi-nel/robutsuri.git
```

応答行列法を用いた拡散方程式の解法

1. 着目メッシュ i に対する流入中性子流 $j^{-}_{i\pm1/2}$ を既知とする
2. 以下の式を用いてメッシュ i における平均中性子束ϕ_iを計算する

$$\left\{\frac{4D}{\Delta x}+\left(1+\frac{4D}{\Delta x}\right)\Sigma_{a,i}\right\}\phi_i = \frac{2D}{\Delta x}\left(4J^{-}_{i+1/2}+4J^{-}_{i-1/2}\right)+\left(1+\frac{4D}{\Delta x}\right)S_i$$

3. 流入中性子流 $j^{-}_{i\pm1/2}$ とノード平均中性子束 ϕ_i から、境界における中性子流 $J_{i\pm1/2}$ を以下の式で計算する

$$J_{i\pm1/2}=\frac{-\frac{2D}{\Delta x}\left(J^{-}_{i\pm1/2}-\phi_i\right)}{2a}$$

4. 流入中性子流 $j^{-}_{i\pm1/2}$ と中性子流 $J_{i\pm1/2}$ から、放出中性子流 $j^{+}_{i\pm1/2}$ を以下の式で計算する

$$J^{+}_{i\pm1/2} = J_{i\pm1/2} + J^{-}_{i\pm1/2}$$

中性子束を
Jupyter Notebook で可視化

２群問題への拡張

SOR法の実装方針

- n回目の外側反復の中性子束 ϕ_i^n を n−1回目の外側反復 ϕ_i^{n-1} を用いて次式により外挿する

$$\phi'^{n}_{g,i} = \phi^{n}_{g,i} + \omega\left(\phi^{n}_{g,i} - \phi^{n-1}_{g,i}\right)$$

- 必要な基本的な処理
  - 外側反復において、前回の中性子束を保存しておく
  - 今回の中性子束を求める際に、上式により中性子束を外挿する

演習問題

- ２群の反射体ありの体系において、反射体厚さと実効増倍率の関係を図示せよ
  - 反射体厚さ：０～50cm
  - 無限反射体厚さ（＝実効増倍率の変化が顕著でなくなる厚さ）

補足資料

オブジェクト指向と関数指向の対比

項目	オブジェクト指向	関数指向
データの所在 (一時変数やヒープ領域は除く)	オブジェクト内部	メイン関数の内部
機能の実現	クラスにより実現	関数やモジュールにて実現
データの受け渡し	「実体」もしくは「参照」渡し	「実体」もしくは「ポインタ」渡し
データの抽象化	クラスを用いて積極的に行う	構造体を用いた手法は可能
制御の流れ	メソッド呼び出しの流れ	関数呼び出しの流れ

オブジェクト指向のメリット・デメリット（※）

特徴・手段	メリット	デメリット
データ隠蔽	データの独立性が高いため、保守性・拡張性・柔軟性に優れる	メソッド経由のアクセスにオーバーヘッドが生じる
データ抽象化	基本型と同様に扱うことができる	データ操作の直交性を確保しようとすると複雑化する傾向にある
クラス継承	差分プログラミングにより実装を簡素化できる	継承関係が複雑化すると管理が難しくなる
ポリモーフィズム（多相性）	複雑な動作や動的な変化を容易に実装できる	挙動の静的分析やデバッグが困難になる

本講義では議論しないので、各自調べるように！