Robutsuri Programming DOJO

㈱原子力エンジニアリング
巽　雅洋

1. プロローグ

今回、なぜプログラミングがテーマの一つになったのか･･･。そして、何故、講師として登壇することになってしまったのか･･･。ま、読者にとってはどうでも良いことかも知れないが、一応まとめておきたい。
時は2017年秋。北海道大学工学部で日本原子力学会秋の大会が開催され、筆者も参加していた。現地委員をされていた某CB先生は忙しそうにしていたが、少し時間がとれそうということで、とある教室にて雑談をしていた。そんな時、困ったような顔をして、つぎのようなネタフリがあった。

次の炉物理夏期セミナーは北大が幹事なんですが、何か良いアイデアは無いですかね～？

少し前にあった第49回炉物理夏期セミナーでも講師を~~させられた~~させて頂いたのだが、その時にあったワークショップ的な取り組みがなかなか良い印象だった。そんなわけで、ワークショップ形式にすれば、もっと踏み込んだことが出来るのではないか･･･と漠然と思っていたこともあり、ついつい後先考えず、こんなことを口走ってしまった。

そうですね～。どうせなら、ワークショップ形式で、実践しながら学ぶとかいうのはどうでしょう？例えばプログラミングとか･･･

また、最近の学生や新人は、Excel などのソフトはある程度使いこなすけれども、ある一定の規模を超えたデータ処理をやらせようとすると急に困り果てる･･･そんな傾向があるようにも感じていた。それってもしかして、プログラミングが出来ないからのではないか！？　･･･と、そんな思いもあった。

そんなこともあり、以前からもやもやと頭の中にあった「プログラミングは最強スキル。皆が持てばもっと生産性が上がるはず！」という気持ちから、先の発言が出てしまったのだ。

しかしながら、CB先生はそのその瞬間を逃さず、目をキラリと光らせたような気がした。

プログラミング、それいいですね！　夏期セミナーは第50回目の記念すべき回ということで、ぜひお願いします。
ね、巽さん！

しまった!?

CB先生の持ち前の~~強引さ~~リーダーシップで、あっという間に~~丸め込まれた~~話がまとまったのであった。めでたしめでたし。

･･･と、一部脚色はあるがこのような経緯で、過去の炉物理夏期セミナーで初めての試みであると思われる、プログラミング演習をすることになった次第である。今から思えば、前回の講義資料の最終頁に、こうなることがあらかじめ織り込まれていたのかもしれない･･･。

信じるか信じないかは、あなた次第です！

1.1. 期待される効果・効能

本 DOJO を修了した暁には、以下のような明るい未来があなたを待っているだろう。

- オブジェクト指向プログラミングの考え方とコーディングの基礎知識が身についている
- 固有値問題に対する数値解法をプログラミングできる
- 計算結果を速やかに可視化して理解を深めることができる
- ○○さん、プログラミングできてカッコいい！ステキ‼ と言われるかもしれない

ただし、過度な期待は禁物である。

1.2. 対象者と前提条件

炉物理をテーマにプログラミングを進めていくので、拡散方程式に対する基本的な知識を持っていることを想定している。ただし、理論面については（C先生が準備した講義資料で）別途補足するので、きっちりと理解しておいてほしい。

一応、これまでにプログラミングを行ったことが無い人でも理解できるように工夫したつもりだが、もしかしたら理解しにくい部分もあるかもしれない。そんな時には、まずはインターネットで検索して調べてみよう。Youtube や Qiita.com など、情報サイトはいくらでもある。

プログラマー界隈ではこういう言葉がある。「ググレ、カス！」　おっと、いきなり下品な言葉で申し訳ないが、くだらない質問をする人ほど、何も調べずにいきなり聞いてくる傾向にある。しかし、これではいくらたってもスキルは伸びない。であるから、あえて言おう。カスであると！　（スマン）

ということで、分からないことがあれば、それが自己成長の一歩なんだと思い、まずはググってみて欲しい。そして、答えを見つけたら情報提供者に感謝し、こんどは自分が情報提供者側に回ってみてほしい。そうすることで、世の中はもっと良くなるはずだ。

1.3. 基本的な考え方

本講義では、プログラミングに関する普遍的なスキルを炉物理をテーマとして身につけてもらうことを目的としている。とはいえ、いわゆるハイパフォーマンス・コンピューティングは目指さない。

スパコンや計算機クラスタを用いるような科学技術計算では、とにかく計算速度が速いことが第一優先とされることが多い。そのために、計算機の能力を最大限に発揮できるようなプログラミングの仕方が求められる。

たとえば、レジスタやキャッシュ利用の最適化、メモリアクセスパターンの最適化、通信方法やタイミングの最適化などなど、それこそソースコードがコンパイルされたらどのようなマシン語に変換されるかとか、実行時にCPUはどのように動くか･･･といったレベルで想像しながらコーディングする必要がある。これは、はっきり言って超マニアックな世界であり、そのような（廃人レベルの）スキルは普通の人には不要である。（が、個人的にはこういうネタは大好きである。そういえば、若かりし頃に後輩には「CPUの気持ちになれ！」とか無茶な指導をしていたなぁ･･･（遠い目） しかし、どのようにしてコンピュータが動いているかについては、できれば興味をもって調べてみて欲しいと願う。）

上の例は極端としても、やはりプログラミングといえば、配列にデータを入れてそれを操作する・・・といったイメージを持っている人もいるかもしれない。確かにそういうプログラミングの方法もある。たとえば、Fortran や C で普通にプログラミングすると、いわゆる関数指向の書き方になって、関数（サブルーチン）に配列へのポインタを渡して・・・といったお作法になる。しかしながら、この方法だとプログラミング初心者にはとっつきにくいし、また、大規模なプログラムを書くことが極端に難しくなってしまう。

したがって、本講義では「速さ」ではなく「早さ」を目指す。

ここはものすごく大事なので、もう一回言っておこう。

「速さ」ではなく「早さ」である。

つまり、計算機の性能を極限まで引き出すことを目的とするのではなく、**短時間に正確に目的を達成するかに重点を置く**ことにする。

そのためには、期待通りに動作する（信頼性が高い）誰にとっても理解しやすい（可読性の高い）プログラムを、できるだけ効率よく（手戻りを最小に）、自信を持って(テストによる裏付けを持って)書く必要があるわけで、そういったことを達成するための実践的な方法論を伝授したい。

筆者の中に暗黙知として蓄積されているノウハウをどこまで整理してお伝えできるか、ある意味筆者自身のチャレンジなのでもある。

というわけで、前置きが長かったが、次章からいよいよ本題に入っていくことにしよう。

2. Part 1:プログラミングとは（理論編）

Wikipedia の「プログラミング」の項目には、は次のように書かれている[1]。

> コンピュータのプログラミング(英：Programming)とは、コンピュータプログラムを作成する事により、人間の意図した処理を行うようにコンピュータに指示を与える行為。まず、そのプログラムの目的、さらには「本当に解決したい問題は何なのか」ということについて十分な検討が必要である。

前半部分の説明はすんなりと入ってくるが、後半部分の説明はなかなか哲学的である。「本当に解決したい問題は何なのか」を理解していないことなど無いだろう･･･と読者は思うかもしれないが、実は「本当に解決したい問題が何なのか」を理解しないまま開発が行われたシステムの例は山ほどある。特に開発規模が大きくなるにつれてこのリスクが高まる一方だ。

では一体全体なぜこのようなことが起こってしまうのだろうか？　どのようにすればこのリスクを低減できるだろうか？　それを理解するためには、少々回り道をしてしまうかもしれないが、このミスマッチ問題の歴史的背景について見ていくことにしよう。

2.1. なぜプログラミングするのか？（目的）

これまた哲学的な問いかけで申し訳ないが、プログラミングを行う目的には次の３つがあると筆者は考える。

- 問題を解決するため
- 問題を理解するため
- プロセスを楽しむため

それぞれについて、もう少し深掘りしていこう。ただし、最後のものについては本講義で対象としないので、参考文献[2] を読んで考察してみて欲しい。

2.1.1. 問題を解決するためのプログラミング

「問題を解決する」はプログラミングの最も主たる目的であろう。コンピュータという決められた手続きを高速かつ正確に処理できる機械を用いて、何らかの問題を自動的に解決したいというのが最も大きなニーズではないだろうか。

であればなおのこと、「本当に解決したい問題」は自分の解決したいニーズに決まってるじゃないか･･･とそう思うかもしれないが、ところがどっこい、ここには大きな落とし穴が待っている場合が多いのだ。

例えばこんな場合を考えてみよう。

いま、多くの手続きが必要な複雑な処理内容について、なんとか自動化したいと思っているとする。そして、そのためにプログラミングを行うべきだと考えていたとしよう。そこにあるニーズは「それぞれの手続きを間違いなく高速に実施すること」と考えがちである。

しかし、**本当に解決すべき問題はそこにあるのだろうか？** すこし考えてみて欲しい。その手続き自体をもっと合理的に変更できないだろうか？　あるいは、そもそもそのような手続きが不要な仕組みを作るということでは無いだろうか？

真の意味でプログラミングを行うということは、単なる高速化を行うということではない。問題を俯瞰し、本来あるべきことはどういうことなのか？について考え、それをシステムとして実現するという作

業なのだ。ある意味、最も適切なゴールを見つける、あるいは定義する作業であるとも言える。

この問題の定義と最適化の作業を「設計」(Software Design)と呼び、それをプログラムコードとして具現化する作業を「実装」(Implementation) と呼ぶ。一般的には、両者は別のスキルであるとされている。特に日本のソフトウェア業界では、「設計」をするのが偉い人（SE）で「実装」は下請けのプログラマー(PG)にやらせる･･･というような風潮がある。だから日本のソフトウェア業界は欧米に負けるのだ･･･と筆者は考えている。

本来、適切に実装出来るのは設計の意図をきちんと捉えられるからであり、逆に、適切な設計ができるのはコンピュータの仕組みや動作を実装レベルで理解できるからである。つまり、設計と実装は表裏一体、陰と陽の関係なのである。どっちが偉いとかそういう問題では無く、どちらも重要なのだ。

本来、プログラミングとは広い概念であるのだが、なぜか「実装」部分、とりわけプログラムを書く作業だけを捉えられることが多い。実際には、ソフトウェア・エンジニアリング（Software Engineering) という言い方が適切ではないだろうか。いずれにせよ、プログラミングとは「本当に解決したい問題は何なのか」を明らかにする作業でもあり、その具体的な方法をコンピュータが理解できる内容に適切に変換していく作業であると言える。特に、問題を解決するためのプログラミングにおいては、いかに自分達が求めるゴールを適切に設定できるかが、そのプログラミングが成功するかどうかを決めると言っても過言ではない。

2.1.2. 問題を理解するためのプログラミング

一方で、最終結果に対するゴールを自分達で適切に設定できない場合もある。

たとえば、一つひとつの要素については十分理解できるが、それぞれが複雑に作用する場合、どのような挙動となるかが容易に予測出来ない問題などである。（例えば、分子動力学に基づく巨大タンパク質の挙動解析、重力多体系における銀河衝突シミュレーション、など）

このように、様々な要素が複雑に絡み合った場合、どのような結果を生み出すか簡単に予測できない事象など、問題自体を理解するための手段としてもプログラミングは有効である。したがって、プログラミングは理論、実験に次ぐ「第三の科学」とも呼ばれるコンピュータ・シミュレーションにおける重要な道具の一つであると言えるだろう。

我々が対象とする炉物理計算も、元々はこの部類に入ると言えるだろう。特にモンテカルロ計算はこの好例である。そもそもモンテカルロ法とは、元々は中性子が物質中を動き回る様子を探るためにスタニスワフ・ウラムが考案し、ジョン・フォン・ノイマンにより命名された手法と言われている。[3]

このように、取り扱いたい問題の概念モデルを作成し、それをコンピュータが取り合え使える数学的モデルや数値計算モデルに落とし込むという、モデリング＆シミュレーション (Modelling & Simulation)という考え方が広く一般的になってきている。

この際に、意図したとおりに動作するかを確認すること（検証）や、期待される結果が得られることを確認すること（妥当性確認）を適切に行うことが大変重要である。ただし、妥当性確認については、直接的に行えない場合が多い。というのも、そもそもよく分からない現象を理解するためにシミュレーションを行うのであって、予測が正しいかどうかを直接的に確かめることが出来ないのが普通だからである。例えば、シビアアクシデントに関するシミュレーションなどはその端的な例である。このような場合、小規模な体系や限られた条件下において妥当性確認をせざるをえない。しかしながら、大規模体系と小規模体系でのスケーリング則 (Scaling Law)の適用可能性や、限定的な条件から一般化に対する説明性の観点から、妥当性確認の実施方法や結果の解釈には注意が必要である[4]。

検証(Verification)と妥当性確認(Validation)は合わせて V&V と喚ばれ、シミュレーションやそれを実現するためのソフトウェアの信頼性を担保するために重要な役割を担っているが、本講義での範囲を大幅に超

えてしまうため、これ以上の議論は割愛する。興味のある読者は「モデリング＆シミュレーション」「Verification & Validation」などでググってみて欲しい。

2.2. どのようにプログラミングするのか？（手段）

プログラミングの目的が明確になったとして、実際にどのような方法でプログラミングを行えば良いのだろうか？

これは壮大なテーマであり、一言で説明するのは難しい。筆者の限られた知識と経験に基づくことになるが、なんとか書き下してみたいと思う。

2.3. オブジェクト指向のすすめ

筆者がオブジェクト指向という概念に出会ったのは、かれこれ20年以上も前のことである。Smalltalk や Object Pascal について書籍を通じて学んだが、実際のプログラミングを体験したのは C++ が初めてであった。（その後、後に MacOS X のベースとなった NeXTStep 上で Objective-C を体験）

当時、X68000 という SHARP製パソコンで稼働する Ko-Window System というウィンドウシステムがあった。(ちなみに、当時は高嶺の花であった UNIXワークステーション上で動作する、X-Window System を参考にして作られたものである。これと同じ世界観を楽しめるという意味で画期的だった Ko-Window は一部のマニアの間で大流行した）筆者は Ko-Window 上で動作するとあるアプリケーションをC言語で製作していたのだが、その先輩が C++ のクラスライブラリも作っていたので試しに使ってみた。するとどうだろう、これまで大変だったウィンドウや各コンポーネントの管理が劇的に簡単になったのだった。これは大変な衝撃であった。

これまでの考え方と180度異なると言っても過言では無いオブジェクト指向の考え方は、まさしくコロンブスの卵的な発想であった。そして、いったんオブジェクト指向の考え方に慣れ親しむと、これまでは一体全体どのようにしてプログラムを書いてきたのだろうか･･･？と思えてしまうほど、自然な考え方になってきたのであった。

これまでのプログラムの作り方は、たとえて言うならば、設計者は全知全能の神となる必要があった。細部のロジックや個々の変数、関数の定義やその副作用（関数戻り値以外の影響）について、事細かく把握しておく必要があったからである。しかし、オブジェクト指向ではプログラムを複数のコンポーネントに分割し、互いの内部情報は原則的に参照できない。その代わり、それぞれがメッセージをやりとりしながら協調して動作する。こうして、ボトムアップ的なアプローチと各階層における適切な抽象化により、システム全体を容易に把握することが可能となったのである。

実は、オブジェクト指向の概念は身の回りにたくさん見つけることが出来る。というよりは、オブジェクト指向の考え方自体が、世の中の構造をモデリングした結果であると言えるだろう。

たとえば、ゲームのなかのキャラクターを例に考えてみよう。ここでは、往年の名作であるドラゴンクエスト（通称ドラクエ）のスライムを例にあげてみる。ゲーム中にはたくさんのスライムが登場するが、それぞれの個体は異なる状態を持っている。ただし、スライムという種類は同じなので基本的な特性は同じだから、動作もひとくくりに定義できる。

この定義全体のことをオブジェクト指向ではクラス(Class)と呼び、各個別の動作に関する関数定義のことをメソッド (Method)と呼ぶ。また、異なる状態を持つそれぞれの個体のことをインスタンス (Instance)やオブジェクト(Object)と呼んでいる。（オブジェクトという呼び方は、オブジェクト指向と紛らわしいので、以後はインスタンスで統一する）

それぞれのスライム・インスタンスは、保持する内容は異なるが、処理手続きはすべて同じだ。たとえば、スライム・クラスにて、「生まれる」「移動する」「闘う」「防御する」「ダメージを受ける」

「死ぬ」などの基本動作（メソッド）を定義しておけば、スライムインスタンスを生成（「生まれる」というメソッドを呼び出す）した後は、それぞれのメソッドを呼び出せば、期待される動作をするだろう。この時、各メソッドの呼び出し元はメソッドの動作がどのように実現されているかは関知しない。また、それぞれのスライム・インスタンスの内部状態(インスタンス変数)は、外部からは見えないように隠蔽されている。外部から内部変数を確認する場合は、それを取得するためのメソッド(getter メソッド）を介して得るというのが基本的な考え方である。

内部変数を自由に参照できないなんて、なんてオブジェクト指向は面倒なんだ･･･と思ったのであれば、それは早計だ。この、メソッドのインターフェース（インプット、アウトプット）をきちんと定義・公開することによって、内部の状態や実現方法に依存することがなくなるのである。つまり、互いの独立性が高まることで、複雑な処理を行う大規模システムを高い保守性を維持したまま実現することが容易になる･･･というわけだ。

もし仮に、それぞれの状態についても明示的に把握・管理ができたとうしよう。そんなとき、一部のモジュールの内部状態の管理方法を変更する必要がでたとしたら･･･一体全体どうなるだろうか？　このようなことは、複数人が関わるプロジェクトでは簡単に起こり得ることだろう。うまく行けば、コンパイルが出来ないとか動作しないという、明らかなエラー状態を生み出すことができる。しかし一番厄介なのは、一見問題なく動いているように見えるが、特定の条件のときにエラーが起こるという、非常に再現性に乏しいバグが発生する場合なのだ。こんな状況のデバッグ作業は本当に大変だろう。想像するだけでもゾッとしてしまう。

こういった非オブジェクト指向なアプローチでは、管理すべき対象が増えれば増えるほど、内部構造が非常になれば複雑になるほど各パートの結合度が高まり、互いに影響を与える可能性が高くなってしまうのである。確かに、関数定義を行い、構造化プログラミングを適切に行うことで、ある程度の規模までであれば問題の発生を抑制することができるが、その分だけ管理コストは増加していく。筆者の体験で言うと、ソースコードが1万行を超える規模あたりから管理コストは飛躍的に高まっていくだろう。

他にも、オブジェクト指向アプローチにおいては、継承(inheritance) や多相性(Polymorphism) という重要な概念がある。どちらもオブジェクト指向のメリットを享受するためには避けて通れない内容であるが、時間の関係から本講義では取り扱わない。興味のある読者はオブジェクト指向プログラミングに関する書籍等を読んでみて欲しい。

2.4. モデリングのすすめ

どんな世界でも、成功するものと失敗するものがある。偶然による場合もあるが、大抵の場合、それぞれにはそれなりの理由があるものだ。このため、成功する確率を高めるためには、過去に成功した事例を研究し、その中から成功に至った要因を分析し、それを再現すれば良い。この基本的な戦略によって実践されるのがモデリング (Modeling) であり、あらゆる分野で基本的なメソッドとして活用されている。

ソフトウェア開発においても、このモデリングが非常に有効である。ここでは、モデリングすべき大きな二つの概念を紹介しよう。

2.4.1. デザインパターンとアンチパターン

前節において、プログラミングとは「本当に解決したい問題は何なのか」を明らかにする作業でもあり、その具体的な方法をコンピュータが理解できる内容に適切に変換していく作業であると述べた。

この作業において、有効な手段として構造化プログラミングといった基本的な考え方はあったが、それほど体系化されたものはなかった。その後、オブジェクト指向アプローチが一般的になるにつれて、成功するソフトウェア開発には一定のパターンがあることを見いだした人々がいた。彼らは、それを「デ

ザインパターン」と呼び、ソフトウェア設計のノウハウを再利用しやすいようにカタログ化したのである。[5]

もともと、「デザインパターン」は建築の世界での概念であったが、それがソフトウェア開発の世界に導入され、開発者間でのコミュニケーションの円滑化に寄与した。最初に提唱されたデザインパターンは23種類であったが、近年はそれ以外のパターンも提唱されている。ただし、それぞれのパターンについて熟知しないとプログラムが書けないかというと、まったくそういうことではない。実際、デザインパターンを知らなくても良いプログラムは書けるだろうが、知らず知らずのうちにこれらのパターンを再発明していることがある。結局は、みな同じようなところに考えが行き着くからだろう。

ソフトウェアのデザインパターンを学ぶことは、言ってみれば武道での型を学ぶようなものだ。確立されたパターンにはムダがなく、その効用も証明済みだ。だったら学ばない理由はないだろう。また、デザインパターンに対する共通認識を持つことで、開発者間のコミュニケーション・コストを下げるという副次的な効果もある。「よし、ここは Factory Method でいこう」といえば、それで全ては話が通ったりするからである。

一方、同じパターンでも「やってはいけないパターン」もたくさん存在していて、それらを総称してアンチパターンと呼んでいる。アンチパターンには、ソフトウェア設計・実装からプロジェクト・マネジメントまで、多種多様のアンチパターンが提唱されている。いわば、ソフトウェア開発の「失敗学」だ。これらを学んでおくことによって、将来のリスクを大幅に低減することができるが、意外にアンチパターンを積極的に学ぼうという人はいないようである。既に多くの書籍や論文、Webサイトでの解説もあるので、ぜひとも調べてみて欲しい。

2.5. アジャイル開発

アジャイルソフトウェア開発 (agile software development)[6]、あるいは単純にアジャイル開発という言葉は聞いたことがあるだろうか？　この概念を説明する前に、まずはその対局にあるウォーターフォール開発 (Water-fall development) [7] について説明しよう。

これはコンピュータが発明されてから長らく使われてきた旧来の開発モデルであり、特に大規模プロジェクトにおいては、ほぼ全てにおいてこの開発モデルが採用されていたといっても過言ではないだろう。ウォーターフォール開発モデルでは、プロジェクトの各段階は、要件定義→設計→実装→テスト→運用、といった具合にそれぞれのステージが明確に分離されている。このため、それぞれのステージが完了すると、基本的には前のステージには戻ることは容易ではない。このため、上流側の設計フェーズでは考慮すべき内容が多くて難易度が高いため、経験豊富なエンジニアがこれを担当する場合が多い。しかし、所詮は人間である。予見できないことなんて山ほどある。であるから、実装フェーズに入ってから、実は要件定義の段階で仮定が間違っていた（どっかーん）･･･なんて言うことも良くある（本当はあってはいけないのだが･･･）　もう既にプロジェクトは前に進んでいるから根本的な対応出来ない場合もしょっちゅうである。であるので、「これは仕様です」とか「この不具合に対する改善処理を実施しました」といった~~言い訳やとりあえずの対策~~説明やパッチによるアップデートが延々と繰り返されるのである。

こういった反省から、1990年代後半から欧米を中心にアジャイル開発のムーブメントが顕著になり、やがて日本にもその勢いが伝わってきた。アジャイルソフトウェア開発モデルを一言で言うと、反復（イテレーション）という短い時間単位を採用して、PDCA (Plan-Do-Check-Action) を高速に回しながら開発を進めていく方法である。システム全体を一気に作り込むのではなく、機能単位で開発を進めていき、頻繁にリリースを行っていく。これにより、常に自分達が正しい位置にいるのか、そして正しい方向に向かっているのを確認でき、プロジェクトが失敗するリスクを大幅に低減することができる。現在では、殆どの開発現場でアジャイル開発モデルは採用されている。また、アジャイルの考え方を運用面にまで展開した DevOps (Development & Operation) というモデルに発展している。（興味のある読者はぜひ調べて欲しい）

話をアジャイル開発に戻そう。最も有名な手法は、ケント・ベックらにより提唱されたエクストリーム・プログラミング (XP: Extreme Programming) [8][9] だろう。 XP は小規模・精鋭の開発チームで実践した際に非常に高いパフォーマンスを発揮する方法で、5つの価値と19の具体的なプラクティス（実践）が定義されている。ドキュメントよりもソースコードを重視し、組織的開発の歯車となることよりも個人の責任と勇気を重んじる人間中心の開発プロセスであるとしている。

XP で定義されている開発のプラクティスについては、Wikipedia の解説が良くまとまっているので~~パクって~~引用し、筆者の経験に基づく解説を付記しておく。（少なからず個人的なバイアスが入っているのでその点は考慮して読んでいただきたい。）

テスト駆動型開発 (TDD: Test-Driven Development)

> 実装を行うより先に、テストを作成する。実装は、そのテストをパスすることを目標に行う。テストを先に作成することで、求める機能が明確化され、シンプルな設計が可能になる。なお、このテストは、部品単位での正確性を確認するユニットテスト（ホワイトボックステスト）と、全体が顧客の要求を満たしているかを確認する受け入れテスト（ブラックボックステスト）の観点で作成する。テストは、自動テストであることが推奨される。なぜなら、それによって、コードの共同所有、リファクタリング、頻繁な結合が可能になるため、開発が進んでも変更コストの増大を抑制することができる。

最初に TDDの概念を知ったときに、「最初にテストコードを書くなんて、それはないわ〜。テストコード書くの面倒やし」と正直思った。でも新しい概念を知ったら、やっぱり試してみたいものである。で、試しにやってみたら、これが意外に楽しかった。そう、テストが楽しかったのである。こんな感覚は、この時が初めてであった。

なんと言っても、テストが通る瞬間はやっぱり快感だ。テストに成功した瞬間、エクスタシーを感じてドーパミンが激しく分泌される。この快感をもっと得ようと、さらにテストを書き、本体を実装していく･･･。こんな風に書くとヤバイ人のように思うかもしれないが、この正のフィードバックは想像していた以上に上手く機能したのである。

本講義でも TDD を実践していく。もしあなたが少しでもエクスタシーを感じられたのなら、開発者としての素養は十二分にあるだろう。

ペア・プログラミング

> プログラミングは、二人一組で行う。一人がコードを書き、もう一人はそれをチェックしながら、仕様書を確認したり全体構造を考えたりするなど、ナビゲートを行う。二人は、この役割を定期的に交代しながら仕事を進める。ナビゲータのサポートにより、以下の効果が得られる。
>
> - 細々とした問題の解決に要する時間が短くなる。
> - 常にコードレビューを行うことができる
> - 集中力が持続する。
> - コードの詳細を理解したメンバーが常に2人以上いることで後々のコード共有に役立つ。

ペアプログラミングの概念も初めて知ったとき、「そんなんしたら開発効率が劇下がりと違う？」と思った。浅はかだった･･･。

実際には、ペアプロは良いことづくめだ。だが、これを本気で実践すれば、非常に疲れる。一日やれば、夕方には疲労困憊･･･といった感じなのである。というのも、横にレビュアーが常にいるわけだから、自分が書くコードに対して常に説明できる状態を維持しなければならない。要するに、いい加減なことが出来ないのだ。これを実践するためには、常に高い集中力を維持しなければならないので、どう

やっても90分が限界だ。途中で休憩を入れて役割を交代しつつ、一日に数セットやれば本当にクタクタになってしまう。

成果が上がるペアプロであるが、大きな欠点が二つある。一つ目は、**自分達は真剣に開発をおこなっているのに、わいわいと楽しそうにしていると、なんだか遊んでいるように見える**という点だ。これは良くコミュニケーションが取れている証拠なのだが、体験したことの無い管理者にはそのように見えてしまうのかもしれない。もう一つは、**開発コストを単純な人工数で管理すると生産性が下がって見える場合がある**ということだ。本来、トータルコストは 人工数÷品質 に比例するものだ。したがって、品質を高めることによりトータルコストを下げることができるのだが、多くの場合はこういった発想が希薄である。これらが日本の開発現場でペアプロが実践されない（実践できない）最も大きい理由なのでは無いだろうか。筆者の経験では、ペアプロを適切に実践することで確実にコードの品質が高まるし、技術共有も促進される。特にベテランと新人を組み合わせたペアプロはとても有効であり、いま叫ばれている技術伝承の問題も一挙に解決できるだろう。であるので、副次的なメリットも含めると、ペアプロはとても有効な手段だと言える。読者がもし開発現場の担当レベルであれば、ペアプロの実践を上申してみよう。リーダー的立場であれば、ぜひとも現場での実践を促してみて欲しい。

ソースコードの共有

> 誰が作ったソースコードであっても、開発チーム全員が断りなく修正を行うことができる。ただし、全てのコードに対する責任を、全員が担う。

ソースコードを共有するためには、いわゆるリポジトリ管理が必要となる。この手段としては古くは RCS (Revision Control System)や SVN (Subversion) が使われたが、これから使うのであれば Git [10] であろう。オープンソースの世界では、Git によるリポジトリ共有サービスである Github [11] による公開・共有は、もはやデファクトスタンダードと言っても過言では無いだろう。Github が発明した "Pull request"により、他人のリポジトリに対して改善提案を簡単に行うことが出来るようになり、集合知としての開発スキームが格段に促進されるようになった。学術目的で開発するプロジェクトで、Github にてソースコードを公開している場合もある。特に炉物理分野においては、米国 MIT が積極的であり、OpenMC[12] や OpenMOC[13] が Github にて公開されている。

複数人で開発しない、いわゆる一人プロジェクトにおいても、リポジトリ管理システムを用いることのメリットは多数ある。というより、リポジトリ管理システムを使用しないとまともに開発はできないだろう。プログラムをリポジトリに登録（コミット）しておけば、任意のタイミングでその状態を復元できるようになる。コミット時のログも見られるし、任意のリビジョン間の差分も簡単に確認できる。さらに、リビジョン・ツリーを分岐させて（ブランチ）たり、統合（マージ）したりと、自由自在に管理できるので、これを使わない手はない。また、**「将来の自分は他人である」というポリシーのもと、適切に情報管理を行う**ことによって、「あれって、どうだったっけ？」という疑問に対しても適切に対処できるようになるだろう。

もし、リポジトリ管理システムを用いなければ、バージョン管理は非常に煩雑になるだろう。同じようなソースコードを含んだフォルダが大量に作られて、何がどう違うのか？が把握できなくなり、早期に破綻することだろう。経験者が語るのであるから間違いは無い。

ということで、リポジトリ管理システム（特に Git 及び Github）に関する学習は、決して損しない投資だと言える。是非とも学んでおいて欲しい。

リファクタリング

> 完成済みのコードでも、随時、改善処置を行う。この際、外部から見た動作は変更させずに、内部構造がより見通し良く優れたものになるようにする。テストが作成されていることが前提になる。

リファクタリングは、re-factor-ing というスペルからも分かるように、ソースコードの中から共通する因子 (factor) をくくりだし、まとめる作業である。たとえば、処理Ａのコードが既に実装済みの場合を考える。そこに処理Ｂを実装する際、処理Ａの内容とかなり似通っている場合、経験の浅い開発者は、処理Ａのコードをコピペ（筆者はこの言葉が大嫌いだ）後に、処理Ｂの目的に合うように変更する･･･といったことをしがちである。その結果、オリジナルである処理Ａのクローンのようなコードが大量に存在することになる。しかし、これはアンチパターンの最たるものだ。

もし、処理Ａの実装に問題があったとしたどうだろうか？　問題が発覚して対応を求められた開発者が経緯を知らない（もしくは忘れてしまっていた）場合、まずは処理Ａを修正するだろうが、似通った問題が残っていないかをどこまで調べるだろうか？　不幸にも処理Ｂが処理Ａと全く違う場所（モジュール等）で実装されていれば、問題があるであろう処理Ｂの存在に気づくことは非常に難しい。そして、マーフィーの法則によって、処理Ｂの問題が噴出するのである。これが安全に関わる内容、たとえばロケット、自動車、医療機器などの制御だったとしたら･･･考えただけでも恐ろしい。

であるので、コピペ（何度も言うが筆者はこの言葉が大嫌いだ）をするのでは無く、処理Ａと処理Ｂの共通因子を処理Ｃとして新たに定義してやり、処理Ａや処理Ｂから呼び出すようにすればよい。これがリファクタリングの根本的な考え方である。

また、オブジェクト指向プログラミングの場合、重要なことはオブジェクト間のインターフェースであるので、メソッドの内部実装は基本的には外部から隠蔽されている。その点で、上記のように共通因子で括るという意外にも、より効率的なコードや可読性の高いコードに変更するといったことも、広義のリファクタリングとして捉えられている。

YAGNI

> You Aren't Going to Need It.（必要なことだけ行う）。先を見据えて、前払い的に機能を増やし、実装を複雑化させることは避ける。むしろ無駄な機能があれば削りとり、今必要な機能だけのシンプルな実装に留めておく。これにより、後のイレギュラーな変更に対応しやすいようにする。シンプルで洗練され、安定性の高い機能・コードのまま、同時に将来的な汎用性も高めることは問題ないが、把握を難しくし、不安定化を招く機能・コードは、可能な限り削り落とす。

要するに、「今必要なものに集中しろ」ということだ。システム開発においては、「あれもいるし、これもいる。やっぱり、そっちもだ」といった具合に、最初から大風呂敷を敷きがちである。もちろん、将来的な構想について見通しておく必要はあるが、**今すぐ必要でないものに対して機能に対してあれこれ心配しても仕方が無い**のである。それよりも、いま必要なもにに集中し、可能な限り問題をシンプルにした方がよっぽど良い。

このような考え方は、何もソフトウェア開発が最初ではない。いろんなところで実践されていて、要するに呼び方が違うだけだ。トヨタ生産方式では「Just in time」と思想のもと、カンバン方式により「必要なものを、必要な時に、必要な量だけ生産する」という管理がなされてきた。また、リーン開発手法[14]では「実用最小限の製品(MVP: Minimum Valuable Product)」という形で表現されている。

なお、筆者は検証や問題解決の際に「最小問題セット」を取り扱うことを心掛けているが、このプラクティスの派生版と言えるだろう。

2.6. その他のベストプラクティス

他にもいろんなベストプラクティスがあり、枚挙にいとまが無い。あまたあるプラクティスから、筆者が特に重要視しているものを幾つか挙げておこう。例のごとく Wikipedia等から~~パクって~~引用しておく。

KISSの原則 (KISS principle)

> KISS の原則 (KISS principle) とは、"Keep it simple, stupid" （シンプルにしておけ！この間抜け）、もしくは、"Keep it short and simple" （簡潔に単純にしておけ）という経験的な原則[1]の略語。その意味するところは、設計の単純性（簡潔性）は成功への鍵だということと、不必要な複雑性は避けるべきだということである。

筆者は学生や新人のころ、恩師や先輩から「出来るだけシンプルに考えるということが大切」「作ったコードが正しいか手計算と比較してチェックすべし！」と教わった。言ってみればこれも KISSの原則に照らし合わせた原理原則である。

少し脱線するが、物事を分類する際には、「容易か困難か」という軸と「単純か複雑か」という軸で整理すると分かりやすい。

(1) Easy & Simple	**(2) Easy & Complex**
(3) Difficult & Simple	**(4) Difficult & Complex**

最も優先すべきは(1)を選択する戦略だろう。この次に選択すべきは(2)もしくは(3)であるが、これは目的と場合に応じて変わってくる。(4)を選択しなくてはならない場合でも直接的に(4)を目指すのでは無く、(2)か(3)を経由することによって成功する確率は高められるので、このことは十分に意識しておきたい。

車輪の再発明をしない

> 車輪の再発明（しゃりんのさいはつめい、英: reinventing the wheel）は、車輪を題材にした慣用句であり、世界中で使われている。「広く受け入れられ確立されている技術や解決法を知らずに（または意図的に無視して）、同様のものを再び一から作ること」を意味する。

ソフトウェア開発における「車輪」とは、主に既に開発されているアルゴリズムやソフトウェア群（ライブラリ）のことを指している。たとえば、固有値問題を解くためのソルバーを開発しようとしていたときに、既にその目的に合致したものがあればそれを使った方が効率的だ。もちろん、経験値を上げるために、敢えて車輪を再発明することも時には必要だ。学校などでの教育はその好例だ。

しかしながら、大抵の場合、特にプロダクション・システムを開発する場合は、既に効果が実証されている既存のライブラリを上手く活用することがとても重要だ。というのも、それらを適切に用いることで、製品の品質向上と開発コストの低減を実現できる可能性が格段に高まるからである。したがって、実際の開発行う際、目的に合致したライブラリが既に存在しないか、十分にサーベイをしておくことが望ましい。まだインターネットが発達していなかった頃はライブラリの公開も一般的で無く、また検索エンジンも発達していなかったため、サーベイ自体も非常に難しかった。しかしながら、ライブラリ公開の仕組み(Python における pip 等のパッケージ管理システム)や github.com 等のリポジトリ共有サービスにより、格段に簡単になっている。さらに、検索エンジンも高度に発達しているため、まずは徹底的に検索してみることを推奨する。

名前重要

プログラミング言語 Ruby の作者である Matz こと、まつもとゆきひろ氏は以下のように述べている。[15]

> 適切な名前をつけられると言うことは、その機能が正しく理解されて、設計されているということで、逆にふさわしい名前がつけられないということは、その機能が果たすべき役割を設計者自身も十分理解できていないということなのではないでしょうか。個人的には適切な名前をつけることができた機能については、その設計の8割が完成したと考えても言い過ぎでないことが多いように思います。

「人は見た目が9割」というヒット本があったが、同様に、「プログラムは名前が9割」と言っても過言ではないだろう。クラス名、変数名、メソッド名、はたまたプロジェクト名など、理想的にはその名前から目的・役割・機能が直感的に理解出来るということが望ましい。いや、望ましいどころではない。**そうなるように名前をつける必要がある**のだ。

一つアンチパターンを紹介しよう。それは、実際の処理内容や目的と異なる名前を関数やメソッドにつけるというものだ。（実際にやったらアカンよ）　たとえば、store_data という名前の関数があったとしよう。この名前から「データを格納する」という機能が想起されるが、実際にはデータを削除するという機能をその関数が持っていたとしたらどうだろうか？　まさしくカオスであり、何を信じたら良いか全く分からなくなる･･･。

読者は「そんなばかげたことはしないだろう･･･」と思うかもしれない。実際にここまで酷い例は無いかもしれないが、たとえば "func1" といった関数名とか、"data1" といった変数名とか見たことはないだろうか？　**この関数・変数はいったい何のために存在するのか？** と疑問に思ったことはないだろうか？もしそんな経験があるとしたら、それはアンチパターンを発見したということで、自分は絶対にそのようなことはしない！と深く心に刻み込んでおいて欲しい。（特に、本講義の Part2 で見つけた場合はね･･･)

要するに、**それだけでその存在意義や機能が分かるように名前をつける** ことが大切だということだ。そのためには名前が長くなっても良いのだ。特に最近では、統合開発環境 (IDE: Integrated Development Environment) や高機能エディタが名前の補完機能を有しているため、長い名前でも全く問題なく使えるようになった。ものには限度というものはあるが、やはり適切な名前をつけることを心掛けたい。名前付けのセンスでその開発者の力量が計り知れる･･･という説もあるが、案外、的を外してはいないのかもしれない。

DRY (Don't Repat Yourself) 原則

DRY原則は、Andy HuntとDave Thomasが、著書「達人プログラマ」[16] の中で提唱した原則で、プログラミングに関して守るべきとされている原則の中でも特に重要なものと言っていいだろう。[17]

なかなか一言で伝えるのが難しい概念だが、 IT用語辞典 e-Words の解説が端的に表現していると思うので、~~パク（もうええって）~~ 引用しておこう[18]。

> DRY原則とは、情報システムの構成や構築手法についての原則の一つで、同じ意味や機能を持つ情報やデータを複数の場所に重複して置くことをなるべく避けるべきとする考え方。
>
> システムやソフトウェアを構成する設定や設計に関する情報について述べたもので、複数の場所に同じ情報が置かれていると変更時に整合性が取れなくなる危険性が高まるため、一箇所で管理して必ずそこから参照するようにすべきとする原則である。
>
> 転じて、プログラムコード中で同じ(ような)動作をするコードを何度も書かずに、一度書いたものを再利用するようにすべきとする意味で用いられることもあるが、こちらは本来は“Once and Only Once”(OAOO)原則と呼ばれるものである。 (IT用語辞典 e-Words より引用）

良く OAOO との混同される概念だが、提唱者である Dave Thomas は以下のように語っている。[19]

> Dave Thomas: Most people take DRY to mean you shouldn't duplicate code. That's not its intention. The idea behind DRY is far grander than that. DRY says that every piece of system knowledge should have one authoritative, unambiguous representation. Every piece of knowledge in the development of something should have a single representation. A system's knowledge is far broader than just its code. It refers to database schemas, test plans, the build system, even documentation.

> Given all this knowledge, why should you find one way to represent each feature? The obvious answer is, if you have more than one way to express the same thing, at some point the two or three different representations will most likely fall out of step with each other. Even if they don't, you're guaranteeing yourself the headache of maintaining them in parallel whenever a change occurs. And change will occur. DRY is important if you want flexible and maintainable software.
>
> The problem is: how do you represent all these different pieces of knowledge only once? If it's just code, then you can obviously organize your code so you don't repeat things, with the help of methods and subroutines. But how do you handle things like database schemas? This is where you get into other techniques in the book, like using code generation tools, automatic build systems, and scripting languages. These let you have single, authoritative representations that then generate non-authoritative work products, like code or DDLs (data description languages).
>
> （以下、筆者による超訳） デーブ・トーマス： 殆どの人は DRY をコードを重複させないことだと思っているけど、そういうことじゃ無いんだ。DRYの背景にあるアイデアはもっと大きいものなんだ。 DRYでは、システムに関するすべての要素は、信頼できる明確な一つの形として表現さるべきなんだ。何かを開発しているとき、あらゆる要素について 、それぞれは一つだけの表現形式をとるべきだ。システムの要素ってのは、コードだけじゃなく、もっと広いもんなんだよ。データベースのスキーマ、テスト計画、ビルドシステム、さらにはドキュメントまで指すからね。
>
> そういう状況において、どうして各機能を表現する一つの方法を見つけないといけないのか？ それは明らかなんだけど、あるとき同じ内容を一つ以上の方式で表現できたとすると、二つも三つも異なる表現形式があったら、それぞれが違ったものになってしまうだろうね。たとえそうならなかったとしても、変更される度にそれぞれメンテナンスする必要があるから、頭痛に悩まされ続けることになるだろうね。で、実際にそんなことると･･･。柔軟で保守できるソフトウェアを望むなら、DRYは重要だよね。
>
> いろいろと違っている全ての要素を、どのようにして一回だけで表現するのか？というのが問題なんだ。これがもしコードの話だけだったら、メソッドとかサブルーチンとかを使って、コードを整理して繰り返しを無くすことはできるよね。でもデータベーススキーマみたいな話だったらどうすれば良いのか？ これについては、コード生成ツールや自動ビルドシステム、スクリプト言語とか、この本での他のテクニックの範疇にはいってくる。これらを使えば、コードや DLL(データ記述言語）みたいな第一義的じゃない中間成果物が作れる、信頼できる単一の表現形式を持つことができるんだ。

最後の下りが理解しにくいかもしれないが、DRY原則を高度に実践していけば、もはやコードやデータベーススキーマ等も自動的に作成できるということを意味している。文中にの "authoritative" という単語は、「（開発者が決める）第一義的、根源的なもの、だから信頼できる」という意味で使われている。そして、上位の情報（メタ情報）を活用することによって「システムを作るシステム」を作ることが可能だということだ。この応用として「メタプログラミング」や「ドメイン固有言語 (Domain Specific Language)」が挙げらる。具体例は Web フレームワークである Ruby on Rails [20] の中で数多く見つけることが出来るだろう。興味のある読者はさらに調べてみて欲しい。

設定より規約 (Convention over Conviguration)

> 設定より規約（せっていよりきやく、英: convention over configuration）とは、開発者の決定すべきことを減少させ、単純にするが柔軟性は失わせないというソフトウェア設計パラダイム。
>
> この言葉は本来、開発者が指定しなければならないのはアプリケーションの慣例に従わない点だけだ、ということを意味している。例えば、モデルの中にSaleというクラスがあれば、データベース

> 中の対応する表の名前はデフォルトでsalesである。この規約から逸脱したときだけ、例えばその表を "products_sold" という名前にした場合だけ、その名前を使ったコードを書く必要が生じる。
>
> 使用しているツールが実装した規約が開発者の望む動作と一致していれば、設定ファイルを書く必要もない。実装規約と望みの動作が違っている場合、必要な動作を設定しなければならない。(Wikipediaより引用)

ソフトウェアを設計する際には、運用時に柔軟性を持たせることが望まれる。そのために、システムでは設定ファイルを用意し、その中にシステム変数の値を設定出来る場合が多い。この考え方自体は有益なものであるし妥当だ。ただし、何でもかんでも設定ファイルの中で定義するようにしてしまうと、今度はその管理の方がとても大変になってくる。

そこで、データベースのテーブル名の命名則だったり、ディレクトリ構成だったり、基本的な部分は「規約」という形でルール化しておくのだ。これによってムダな周辺コードを削減でき、開発者は本来集中すべきロジック部分に注力することができる。ただ、なんでも間でも「規約」としてしまうと、後に混乱してしまうだろう。であるので、何らかの形で明文化してく方が良いだろう。

2.7. プログラム開発において考えておきたいこと

ここでは、上記のグッド・プラクティス以外にも、プログラム開発において考えておきたいとについてまとめておこう。プラクティスが「スキル」とすれば、これから話すのは「マインド」的な内容だ。どれも、これまでの開発を通じて筆者自身が感じてきたことだが、それなりの信憑性はあると思うので少し耳を傾けて欲しい。

2.7.1. プログラムのライフサイクル

まず最初に心に入れておきたいことは、プログラムのライフサイクルについてだ。プログラムの「寿命」と言い換えても良いかもしれない。これに関して常に考えておきたいことは、「もし、これを使い続けるとなるならば、どうすべきだろうか？」ということだ。

「よし、いまから○○プロジェクトにおいてプログラムを開発するぞ！」と、最初から気合いの入っているのならば、それほど心配することはないだろう。というのも、それなりに意識してきっちり準備しようとするからだ。しかながら多くの場合、**当初思っていたよりもずっと長くそのプログラムが使われ続けるという状況が起こってしまう**のだ。

たとえば、

- 急場しのぎで作ったチョイプロが、いつの間にか現場での必須ツールになっている
- 以前のプロジェクトで仮に作ったマクロ入りの Excel ワークシートが今も使われ続けている
- プロトタイプとして作ったつもりが、拡張に拡張を重ねて、本番システムとして稼働している

なんて話、どこかで聞いたことはないだろうか？

次節の品質保証の話とも関係するのだが、設計や実装が良くないと後々の運用・保守 (O&M: Operation & Maintainance) のコスト はとても高くついてしまう。さらに、それらが原因で何かしらのミスが発生した場合は、その対処に膨大なコストがかかってしまうことを肝に銘じておかなければならない。

- もし、もっと仕様の変更に耐えられるような設計と実装にしていたら･･･
- もし、誰かが引き継ぐことを前提に実装したりドキュメントを書いていたら･･･
- もし、開発初期の段階で将来のことも見据えた設計方針をとっていたら･･･

何かあってからでは遅いのだ。今まで何の問題も無かったし、これからも問題は起きないだろう･･･という考えは捨てた方が良い。**今まではたまたま問題なかっただけ**だと理解し、そのことに感謝しよう。後

に説明するレバレッジ戦略とも関係あるため、詳細はそちらで述べることにするが、初期の段階で、もちろん許容できる範囲ではあるが、検討やリサーチをとことんやっておくべきである。初期であればあるほど、僅かな投資でも将来に大きなリターンとなって返ってくる可能性があるからだ。

2.7.2. ソフトウェアと解析の品質保証（確認）について

なんだか非常に重たいテーマだが、これまた避けては通れない内容なので、簡単にまとめておきたい。特に、何らかのシステム開発や解析業務に従事しているエンジニアであれば、よく読んで理解して欲しい。なお、一般論として議論すると非常に抽象的になってしまうので、ここでは例として炉心設計コードとそれを用いた解析業務の品質とその確認方法について考えてみよう。

まず、炉心設計コード（ソフトウェア）の品質確認とはなんだろうか？　それは、**コード開発者が、その計算コードが意図したとおりに動作することを確認すること**である。たとえば、指定されたデータが内部で正しく取り扱われているか？　数式の解法は正しくコーディングされているか？　計算結果は意図したものになっているか？　などをユーザーに提供する前に確認しておかなければならない。この確認作業のことを「検証」(verification) と呼ぶ。検証の方法としては、計算コードのテスト（ユニットテスト、機能テストなど）、解析解との比較、断面積が与えられたベンチマーク問題での他コードとの比較、などが挙げられる。

また、炉心設計コードを用いた計算（解析）の品質確認とはなんだろうか？　それは、**ユーザーが、使用する計算コードが目的に合致した性能を有することを確認すること**である。たとえば、この解析条件でも正しい予測結果が得られるだろうか？ このパラメータをこの範囲まで振っても適切な予測結果が得られるだろうか？　などを、あらかじめ問題解決の適用範囲において、解析コードが十分な予測性能を提供することを確認しておかなければならない。この確認作業のことを「妥当性確認」(validation)と呼ぶ。妥当性確認の方法としては、ベンチマーク問題での他の計算コードとの比較、臨界集合体試験の解析、実機炉心の運転履歴との比較、などが挙げられる。

これらは「検証及び妥当性確認」(V&V: Verification and Validation)として、コード開発や解析業務の品質保証（確認）における重要な概念として認識されている。炉物理分野における V&V の方法論はある程度確立されていると筆者は考える[21]が、熱流動や構造解析の分野においては、依然として重要なテーマとして活発に議論されている。また、学会標準[4]が策定されたり、民間規格[22]が制定されている。興味のある読者はぜひとも一度調べてみて欲しい。

2.7.3. レバレッジ戦略の重要性

以上を読んで、もしかしたらこんな風に思っているかもしれない。

「プログラムのライフサイクルとか品質保証って、どっちも重要性なんは理解出来るけど、実際問題そんなに時間もコストもかけられへんやん！　で、どないすりゃええの？」

確かに、時間も予算も限られているので、どのように対策するかは非常に悩ましい問題だ。申し訳ないが、これに対する明確な答えは持ち合わせていない。それでも、一つの考え方が有効だろうと筆者は考える。それは、このジレンマを「将来得られる利得の積分値を最大化するための最適化問題」と捉えることだ。その際に、時間に対するレバレッジという概念が重要になってくる。

そもそもレバレッジとは「てこの原理」を意味する言葉で、少ない資本を元手に大きな利益を上げる方法として、金融の世界で良く用いられる。これを時間に対する投資にあてはめて、「直近の時間に投資することによって、将来に大きな利得を得る」というふうに捉えるのである。上手くいけば、最小の時間投資で最大のリターンが得られるかもしれない。逆に、直近に多大な時間をかけてしまうと、そのコストは将来に得られる利得で十分にカバーできない可能性もある。そういう意味で、最適化問題であるのだ。

もし将来にわたって何度も使うことが分かっていたり、将来にトラブルがあると莫大なコストがかかると予見される場合、その頻度や許容リスクに応じて、どの程度までの投資が許されるかをよく考える必要がある。このレバレッジ戦略についての定量的な議論は難しいが、**将来の利得に対して現在の時間に投資する**という基本戦略さえ持っていれば、将来におけるリスクを大幅に低減することは可能だろう。

2.7.4. 自己投資の重要性

先ほどのレバレッジ戦略が重要なのは、なにも仕事や研究におけるプロジェクトに限ったことではない。それよりもむしろ、あなたの人生という大プロジェクトにおいてこそ重要となってくるなのだ。**特に20〜30代前半における自分への投資は、将来的に大きなリターンとなって返ってくる**だろう。何事においても決してムダなことはなく、全てが将来の自分に繋がってくる。今の自分に可能な限り投資をして欲しい。

ということで、読者にエールを送ったところで Part 1 を終えたいと思う。

2.8. 参考文献

1. https://ja.wikipedia.org/wiki/プログラミング_(コンピュータ)
2. リーナス・トーバルズ, 「それがぼくには楽しかったから 全世界を巻き込んだリナックス革命の真実」, 小学館プロダクション (2001)
3. 長家康展, 「炉物理分野におけるモンテカルロ計算の現状と将来展望」, 日本原子力学会2018年春の年会計算科学技術部会企画セッション (2018) http://csed.sakura.ne.jp/wp-content/uploads/2018/05/AESJ2018Spring_PL1.pdf
4. 日本原子力学会, 「シミュレーションの信頼性確保に関するガイドライン：2015（AESJ-SC-A008：2015）」 (2016)
5. https://ja.wikipedia.org/wiki/デザインパターン_(ソフトウェア)
6. https://ja.wikipedia.org/wiki/アジャイルソフトウェア開発
7. https://ja.wikipedia.org/wiki/ウォーターフォール・モデル
8. https://ja.wikipedia.org/wiki/エクストリーム・プログラミング
9. ケント・ベック、「エクストリームプログラミング」, オーム社(2015)
10. https://git-scm.com/
11. https://github.com/
12. https://github.com/mit-crpg/openmc
13. https://github.com/mit-crpg/OpenMOC
14. アッシュ・マウリャ、「Running Lean ―実践リーンスタートアップ」, オライリージャパン (2012)
15. https://プログラマが知るべき97のこと.com/エッセイ/名前重要/
16. Andrew Hunt, 「新装版 達人プログラマー 職人から名匠への道」, オーム社(2016)
17. https://プログラマが知るべき97のこと.com/エッセイ/DRY原則/

18. http://e-words.jp/w/DRY原則.html

19. https://www.artima.com/intv/dry.html

20. https://rubyonrails.org/

21. 巽雅洋,「軽水炉炉心設計コードに関するV&Vの一例」, 第44回炉物理夏期セミナーテキスト (2012)

22. 原子力安全推進協会,「原子力施設における許認可申請等に係る解析業務の品質向上ガイドライン」(2014)　http://www.genanshin.jp/archive/qualityimprovement/data/jansi-gqa-01.pdf

3. Part 2: 炉物理プログラミング (実践編)

では、いよいよ本題のテーマである炉物理プログラミングを始めてみる。

いきなり実践に入る前に、少しだけ準備運動をしておこう。

事前課題において Python言語の基礎的な知識は習得しているはずだが、少しだけおさらいしておく。

3.1. Python言語について

Python言語は、いま最も使われているスクリプト言語の一つだと言って良いだろう。特に、最近は機械学習・深層学習がとても注目されているが、その関連でも Pythonは非常によく使われている。

また、欧米の高等教育現場において Python の利用率は非常に高い。少し前は MATLAB がメジャーだったが、今は完全に Python が追い越したと言っても良いだろう。このように、Python は教育現場や実プロジェクトで多く用いられているため、是非とも使いこなせるようになっておきたい。

プログラミングのエッセンスはプログラミング言語に依存しない。また、プログラミング言語は互いに影響をうけあいながら発展している。したがって、一つのプログラミング言語をマスターしておけば、他の言語の習得は比較的容易に行える。このことからも、まだ本格的にプログラミングを学習していないのであれば、Python を最初に学習することを推奨する。

3.1.1. 必須文法・構文

Python の文法は非常にシンプルだ。また、構文（スコープ）がインデントにより決定されるという点が特徴で、この部分は賛否両論があるのだが、なれてしまえばそれほど気にはならない。インデントと上手く付き合うためには、「Python モード」があるエディタを活用する方が良いだろう。

変数・演算

これといって特徴はなく、他の言語と同様に扱える。ただし、インクリメント・デクリメント演算子(++, --) は存在しないので C++使いの人は要注意だ。

```
hoge = 10
page = ( hoge + 1 ) / 2.0
page += hoge
```

リスト・辞書・タプル

Pythonでは、配列は リスト (list)で、ハッシュは辞書(dic) という型で実現している。タプル(Tuple)は Python独自の型で、値の変更が不可能なリストと理解すれば良い。リストはブラケット(bracket)、辞書はブレース({})、タプルは括弧(())にて表記する。リストやタプルについては、一部を切り出す「スライシング (slicing)」が可能であり、上手く使うと実装が劇的に簡単になる場合がある。

```
the_list = [1,2,3,4,5,6]

the_first  = the_list[0]     # 1
the_last   = the_list[4]     # 5
the_last2  = the_list[-1]    # 6
the_slice  = the_list[1:4]   # [2,3,4]
the_slice2 = the_list[1:4:2] # [2,4]
```

```
the_dic = {'hoge':1, 'page':2, 'hogepage':3}

hoge = the_dic['hoge']           # 1
the_keys = list(the_dic.keys())  # ['hoge', 'page', 'hogepage']

the_tuple = (1,2,3)
the_first = the_tuple[0]     # 1
the_slise = the_tuple[0:2]   # (1,2)
```

判断

if文の最後のコロン(:)を忘れないようにすること。また、if文の条件部分には括弧は不要である。（括弧をつけても文法上問題はないが、使わないのがPython流）　なお、「後置if」はないのが Rubyist には残念だが、Python のポリシーから考えると仕方がないところだ。

```
if hoge == page:
    print("hoge is the same as page!")
elif hoge > page:
    print("hoge is greater than page!")
else:
    print("hoge is smaller than page!")
```

繰り返し

繰り返しには For構文とWhile構文を知っていたら十分だろう。

```
the_list = [1,2,3,4,5,6]
for i in the_list:
    print(i)

total = 0
idx = 1
while idx < 5:    # done when idx in [1,2,3,4]
    total += the_list[idx]
    idx += 1
print(idx,total)  # 5, 14 == 2+3+4+5

while True:
    if total > 20:
        break     #
    print(total)
    total += 1
```

関数定義

関数定義も簡単である。名前付き引数(named parameter)や省略値を用いることが出来るので便利だ。また、戻り値にリスト、タプル、オブジェクトを用いることで、複雑な戻り値を返すことができる。関数の役割は与えられた引数に対する関数値を返すことである。よってそれ以外の効果を「副作用(side effect)」と呼ぶ。例えば、画面に値を表示したり、データベースに記録したりするなどがある。副作用といっても、決して悪影響を及ぼすことを意味するのではない。

```
def die_risk (weight, food, drink="coke"):
    factor = 1.0
    if food == "salad":
        factor = 0.5
    elif food == "pizza":
        factor = 1.5

    if drink=="coke":
        factor *= 2.0
```

```
    the_risk = (weight-50) * factor
    print("Die risk: ", the_risk)   #       (side effect)

    return the_risk


die_risk(60, "salada", "water")
die_risk(food="pizza", weight=70)
die_risk(food="humberger", drink="coke", weight=100)
```

モジュール

他のプログラムから再利用できるようにしたファイルのことを「モジュール」という。

モジュールには、自分でPythonを用いて作成する「通常モジュール」、あらかじめ装備されている「組み込みモジュール」、C言語など他の言語で開発された「拡張モジュール」に分類される。モジュールをパッケージとしてまとめ、「外部ライブラリ」として公開されているので、どんなライブラリがあってどのように使えるのかを知っているかどうかで効率が劇的に変わる。欲しいライブラリは、大抵の場合、既に誰かによって開発されている。車輪の再発明を行わないためにも、事前に入念な調査が必要だ。

モジュールの読み込みは import 文を用いて行う。

```
import random      #
from math import sin, cos   # math             sin, cos

random.random()    # 0.9065161930701089
sin(0), cos(0)     # (0.0, 1.0)
```

クラス定義

Python におけるクラスの定義は、他の言語とよく似ており、特に難しいところは無いだろう。メソッドの定義において、第一引数が自分自身を現すselfとなる点に注意。(C++のthisに相当） なお、コンストラクタメソッドは __init__ である。 内部変数にアクセスする場合には、かならず self が必要なので注意が必要だ。

worker.py

```
import random

class Worker:

    def __init__(self, initial_energy=10):
        self.energy = initial_energy

    def eat(self, the_energy):
        print("Got energy {:.1f}".format(the_energy))
        self.energy += the_energy

    def sleep(self, duration):
        print("Seeep {:.1f} hours".format(duration))
        self.energy += 5.0 * duration

    def work(self, duration):
        print("Work  {:.1f} hours".format(duration))
        self.energy -= 10.0 * duration

    def get_energy(self):
        return self.energy

    def is_alive(self):
        if self.energy > 0:
            return True
```

```
        else:
            return False


guy = Worker()

ENERGY_UNIT = 10
SLEEP_UNIT = 8
WORK_UNIT  = 8

while True:
    feed_energy = random.random() * ENERGY_UNIT
    guy.eat(feed_energy)

    sleep_duration = random.random() * SLEEP_UNIT
    guy.sleep(sleep_duration)

    work_duration = random.random() * WORK_UNIT
    guy.work(work_duration)

    if guy.is_alive():
        print("   Still Alive: {:.2f}".format(guy.get_energy()))
    else:
        print("   DEAD: {:.2f}".format(guy.get_energy()) )
        break
```

実行例はこちら。なお、乱数を使っているので、実行毎に結果は異なる。これも一種のシミュレーションと言えるか!?

```
Got energy 9.3
Seeep 6.8 hours
Work  4.4 hours
   Still Alive: 9.83
Got energy 4.7
Seeep 7.5 hours
Work  5.5 hours
   DEAD: -2.62
```

3.2. 1群1次元拡散ソルバーの実装

ではいよいよ本題に入ろう。本節では、炉物理計算で最も簡単な問題に該当する、1群1次元拡散方程式を解くプログラムを作っていく。計算コードというほど大げさなものでは無いが、ここでは敢えてそういう言い方をしよう。

さて、一言に1群1次元拡散計算コードといえど、解くための数値解法や実装方法はたくさんある。それらを列挙するのは本講義の本題では無いため、いきなり結論を述べるが、ここでは応答行列法に基づくソルバーを実装していくことにする。

なお、拡散方程式の応答行列法に基づく求解については、夏期セミナーテキストの~~CB~~千葉先生による解説を参照されたい。

3.2.1. Step0: 全体構想を練る

計算コードであれ、何であれ、システムを構築する際には全体構成を考えておく必要がある。この際のポイントを挙げておこう。

1. システムの目的を明確にする
2. システムを幾つかのサブシステムに分割する
3. 上記の 1. と 2. を細分化出来ないところまで実施する
4. サブシステムの内部状態と外部との関わり（インターフェース）を規定する
5. サブシステムと上位システムのインターフェースを規定する
6. システム間のインターフェースが上手く機能するかチェックする

7. マクロ側の視点から、全体フローが問題なく流れそうかをチェックする
8. 納得がいくまで上記ステップを繰り返す

さっぱり意味不明だ･･･という場合は、とりあえずはそんなもんか･･･と思っておけば大丈夫だ。

こういったプロセスを上流工程設計と言うが、実はかなりの経験を積まないと適切な設計を行うことは難しい。というのも、上流工程における検討作業は非常に抽象度が高く、システムのデザインパターン・アンチパターンについての知識や、設計文書を作成する技術を習得する必要があるからだ。特に後者については、UML (Unified Modling Language) と呼ばれる表記法を用いることが多い。これを勉強しておけば、複雑なシステムの設計に対しても、自信を持って対応することが出来るようになるだろう。興味がある読者はぜひ調べてみて欲しい。

さて、本題に戻ろう。

ここでは、1群1次元拡散計算コードのシステム設計を行っていく訳だが、どのように考えれば良いのだろうか？　具体的に考えてみよう。

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という方法が有効**だ。もちろん、この方法が唯一の方法ではないが、人間の思考パターンやオブジェクト指向の考え方に沿っているため、直感的に理解しやすいからだ。

Tips: システム設計で常に意識しておきたいこと

システムを開発する際に重要視すべきことは、大規模になれば大規模になるほど、そのライフサイクルが長くなるという事実だ。これは、保守期間が長くなるということを意味する。そのシステムに自分自身がずっと関わるのであればまだ良いが、多くの場合は将来にわたって複数の人が関与することになる。この際、最も重要視すべきものは、実装中のコメントでも外部のドキュメントでもなく、分かりやすい設計（と実装）自体である。いくらドキュメントが整備され、コメントが詳細に記載されていても、難解な設計だったり「ダメダメな設計」だったとしたら目も当てられない。

ここで言う分かりやすい設計とは、上記で述べたように、直感的に理解できるプログラム構造やオブジェクトクラスへの分割を意味する。あと、忘れてはいけないもう一つの重要なことは、将来の自分は他人と同じであるという事実だ。完璧な記憶を持つ人で無い限り（ほぼ全ての人がそうだろう）、自分が設計した内容であっても時間が経てばその詳細は忘れてしまう。ただし、分かりやすい設計であれば、見た瞬間に理解したり記憶を呼び戻したりするだろう。理想的な設計とは、かくあるべきである。

思いっきり脱線してしまったので、再び本題に戻ろう。

今のテーマは拡散方程式を解いて、実効増倍率や中性子束分布を求めることだ。なので、計算体系を定義して、その媒質中における中性子の拡散現象を取り扱う必要がある。

ということで、システムの設計をする際には、次のような感じで考えを巡らすことだろう･･･。

> 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。
>
> 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。
>
> 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！？
>
> 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！？
>
> 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挙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。
>
> 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!
>
> 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。
>
> 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。
>
> 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？

･･･というような感じで考えただろう（と、いうことにしておこう･･･）

こんな風な思考プロセスを経て、クラス構造をザックリとまとめたのが以下の図だ。

- 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 (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

3.2.2. Step1: 基本的なオブジェクトの設計と実装

基本的な方針が出せたので、いよいよ実際にコーディングを始めてみよう。

ここでは、最も基本的なデータ要素となる断面積クラスを実装してみよう。

cross_section.py

```
import numpy as np

#
N_REACT = 3  # D, Siga, nuSigf
DIF    = 0
SIGA   = 1
NUSIGF = 2

class CrossSection:
    """
            (1  )
    """

    def __init__(self, val=None):
        """

        """
```

```
        self.x = np.zeros(N_REACT)

    def set_d(self, val):
        self.x[DIF] = val

    def set_siga(self, val):
        self.x[SIGA] = val

    def set_nusigf(self, val):
        self.x[NUSIGF] = val

    def dif(self):
        return self.x[DIF]

    def siga(self):
        return self.x[SIGA]

    def nusigf(self):
        return self.x[NUSIGF]

    def debug(self):
        print("-" * 9 + " XS " + "-" * 9)
        print("D\tSiga\tNuSigf")
        print(self.x[DIF], self.x[SIGA], self.x[NUSIGF], sep='\t', end='\n')
        print("-"*22)

if __name__ == '__main__':
    xs = CrossSection()
    xs.set_d(1.0)
    xs.set_siga(2.0)
    xs.set_nusigf(3.0)
    xs.debug()
```

クラス定義やメソッド定義の直下にはコメントを入れることが出来るので、必要に応じて入れると良いだろう。

最初ということで、必要最小限のメソッド群を定義している。具体的には、コンストラクタ(__init__)、拡散係数の設定(set_d)、吸収断面積の設定(set_siga)、生成断面積の設定(set_nusigf)、拡散係数の取得(dif)、吸収断面積の取得(siga)、生成断面積の取得(nusigf)、デバッグ用メソッド(debug)である。最後のデバッグ用メソッドについては、文字列化する特殊メソッド __str__ として定義する方法もある。

ここで、数値演算用ライブラリである Numpy を用いていることに注目して欲しい。Numpy は Python で高速に計算させる場合には必須なので、必ずチェックしておこう。

最後の if ブロックは、このスクリプトを

```
python cross_section.py
```

と実行した際に流れる部分であり、よく使われるテクニックであるので、これも覚えておこう。

さて次に、計算ノードも実装しておこう。こんな感じになる。

node.py

```
import numpy as np
from cross_section import CrossSection
from config import *

class Node:
    def __init__(self, xs=None):
        self.jout = np.ones(2)    # out-going, [XM, XP]
        self.jin  = np.ones(2)    # in-coming, [XM, XP]
        self.flux  = 1.0    # average flux
        self.width = 1.0
        self.xs = None
```

```
        self.keff = 1.0
        self.fis_src = 1.0
        if(xs):
            self.set_xs(xs)

    def set_xs(self, val):
        self.xs = val

    def calc(self):
        pass

    def debug(self):
        print("-"*3 + " Node " + "-"*40)
        print("  jin_XM \t", self.jin[XM] )
        print("  jin_XP \t", self.jin[XP])
        print("  jout_XM\t", self.jout[XM])
        print("  jout_XP\t", self.jout[XP])
        print("  flux   \t", self.flux)
        print("  keff   \t", self.keff)
        self.xs.debug()
        print("-"*50)


if __name__ == '__main__':
    node = Node()
    xs = CrossSection()
    xs.set_d(1.0)
    xs.set_siga(2.0)
    xs.set_nusigf(3.0)
    node.set_xs(xs)
    node.debug()
```

冒頭の import 文で、CrossSection オブジェクトの定義を cross_section.py から読み込んでいる。また、次の config.py で定義されている定数を読み込んでいる。

```
#

##
XM = 0
XP = 1
```

Node では、部分中性子流、中性子束、メッシュ幅、断面積オブジェクト、実効増倍率、核分裂源をそれぞれ定義している。

calcメソッドもとりあえず定義しているが、現時点で実態は無い。とりあえず、メソッドだけを準備しておくということは初期の段階で良くやる方法だ。

3.2.3. Step2: テストコードを準備する

次に、いよいよテストコードを準備する。テスト駆動開発に慣れてくれば、いきなりこのステップから始めても良い（というか、むしろその方が自然）。

ここで問題が出てくる。それは、実際のコード本体とテストコードが混在してしまうということだ。この回答として、それぞれを別々のディレクトリに集約するということを良くやる。たとえばこんな感じだ。

```
(project_root) ----+ (                      )
                   |
                   +--- lib----+--- cross_section.py
                   |           |
                   |           +--- node.py
                   |
                   +--- tests--+--- test_cross_section.py
                               |
                               +--- test_node.py
```

ではさっそく実際のテストコードを見てみよう。

test_cross_section.py

```
import unittest

import sys
sys.path.append('../lib')

from node import Node
from cross_section import CrossSection

class CrossSectionTest(unittest.TestCase):

    def test_sets(self):
        xs = CrossSection()
        xs.set_d(1.0)
        xs.set_siga(2.0)
        xs.set_nusigf(3.0)
        self.assertEqual(xs.dif(), 1.0)
        self.assertEqual(xs.siga(), 2.0)
        self.assertEqual(xs.nusigf(), 3.0)

    def test_sets3(self):
        xs1 = CrossSection()
        xs1.set([1.0, 2.0, 3.0])
        xs1_ref = CrossSection()
        xs1_ref.set(xs1)
        self.assertEqual(xs1, xs1_ref)


if __name__ == '__main__':
    unittest.main()
```

なんとなく「読める」と思うが、いかがだっただろうか？

ここでは、単体テスト・フレームワークのライブラリである unittest を用いた。Python に標準ライブラリとして準備されているので、すぐに使えて便利だ。

unittest では、各テストメソッドは名前が test から始まる規約となっているので、そのような命名ルールになっている。各メソッドでは、CrossSection オブジェクトを生成してから、各断面積の値をセットしている。そして、assertEqual メソッドにて、第一引数と第2引数が等しいかどうかをチェックしている。assert 系メソッドにはいろいろあるので、目的に応じて使い分けると良い。

ところで、cross_section.py や node.py が異なるディレクトリ (../lib) にあるため、import の際の読み込みパスを追加している点に気をつけて欲しい。これを忘れると import 時にエラーとなるので注意が必要だ。

test_set3 メソッドにおいて、CrossSection クラスの set メソッドの呼び出しているが、まだこのメソッドは定義していない。とりあえず、テストを走らせてみると･･･以下のような結果となった。

```
% python test_cross_section.py
.E
======================================================================
ERROR: test_sets3 (__main__.CrossSectionTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_cross_section.py", line 22, in test_sets3
    xs1.set([1.0, 2.0, 3.0])
AttributeError: 'CrossSection' object has no attribute 'set'

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (errors=1)
```

これでいいのだ！　敢えて、テストが失敗することを確認したのだ。このテストが通るように CrossSection クラスにメソッドを追加のが下のものだ。

cross_section.py

```
import numpy as np
import copy

#
N_REACT = 3  # D, Siga, nuSigf
DIF    = 0
SIGA   = 1
NUSIGF = 2


class CrossSection:
    """
                (1  )
    """

    def __init__(self, val=None):
        """

        """
        self.x = np.zeros(N_REACT)
        if(not (val is None)):
            self.set(val)

    def set(self, val):
        if (type(val) == CrossSection):
            self.x = copy.copy(val.x)
        else:
            for k in range(N_REACT):
                self.x[k] = val[k]

    def set_d(self, val):
        self.x[DIF] = val

    def set_siga(self, val):
        self.x[SIGA] = val

    def set_nusigf(self, val):
        self.x[NUSIGF] = val

    def dif(self):
        return self.x[DIF]

    def siga(self):
        return self.x[SIGA]

    def nusigf(self):
        return self.x[NUSIGF]

    def __eq__(self, other):
        return np.allclose(self.x, other.x)


    def debug(self):
        print("-" * 9 + " XS " + "-" * 9)
        print("D\tSiga\tNuSigf")
        print(self.x[DIF], self.x[SIGA], self.x[NUSIGF], sep='\t', end='\n')
        print("-"*22)


if __name__ == '__main__':
    xs = CrossSection()
    xs.set_d(1.0)
    xs.set_siga(2.0)
    xs.set_nusigf(3.0)
    xs.debug()

    xs2 = xs * 2.0
    xs2.debug()
```

実装できたら先ほどのテストを再び実行してみよう。下記のようにテストが成功するはずだ。

```
% python test_cross_section.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

えっ？ なんでこんな面倒くさいことをするのかって？

まぁ、この段階では面倒な部分しか見えないだろう。しかし、テストコードを書くということは、幾つものメリットがあるのだ。そのうちの大きなものとしては、次の3つが挙げられる。

- テストコードの作成を通じて、必要なロジックを素早く書ける
- 常にテストを実行することで、いち早くエラーを検出できる
- テストコードが上位層クラスにおけるメソッドのひな形になる

最初にテストを書く "Test First" の姿勢を取っていれば、木を見て森を見ずという状態にはなりにくい。というのも、**テストコードが「成功」する状態にするために、ライブラリに記述する最小限のコードに集中できる**からだ。これが1番目のメリット。いったんテストコードが成功すれば、さらに、**システムがあるべき姿になったときに成功すべきテストコードを追加する**。そして、次にこのテストが成功するようにライブラリを作っていく･･･。あとはこの繰り返しによって、必要なロジックを素早く書くことができる。

繰り返しテストを実行しているわけであるから、万一エラーが発生したとしても、**テストがもれなく・重複なく作成されている限り**、その段階ですぐに見つかる。これが二番目のメリット。テスト駆動開発の要はテストコードをしっかり書くということに他ならない。そのため、テストコードの行数がライブラリ本体よりもずっと多いというプロジェクトも決して珍しくない。「テストを最初に書いて、それが成功するようにライブラリ本体を書く」とう発想の転換は最初は衝撃的かもしれないが、すぐに慣れてくる。そのうち、テストコードを書いてテストを通すのが快感(?)になってくるはずだ。なぜなら、テストコードはテストのためにあるのでは無く、**クラスの使われ方（ユースケース）を決定するために作成するもの**だからだ。

そして3番目のメリットについては、なかなか言葉だけでは説明が難しいので、後で事例を示して説明することにしよう。

ということで、次に行く前に Node クラスのテストコードも示しておこう。

test_node.py

```
import unittest

import sys
sys.path.append('../lib')
import math

from config import *
from cross_section import CrossSection
from node import Node

class NodeTest(unittest.TestCase):

    def test_onenode(self):
        node = Node()
        xs = CrossSection([1.36, 0.0181, 0.0279])
        #xs.debug()
        node.set_xs(xs)
        node.set_keff( xs.nusigf() / xs.siga() )
        node.calc()

        for k in range(100):
```

```
            jout_xm = node.get_jout(XM)
            jout_xp = node.get_jout(XP)
            node.set_jin(XM, jout_xm)
            node.set_jin(XP, jout_xp)
            node.calc()

        node.debug()

        self.assertEqual(node.get_jout(XM), node.get_jout(XP))
        self.assertEqual(node.get_jin(XM), node.get_jout(XM))
        self.assertEqual(node.get_jin(XP), node.get_jout(XP))
        self.assertEqual(node.get_jout(XM)+node.get_jin(XM), node.get_flux() / 2.0)


if __name__ == '__main__':
    unittest.main()
```

先にも述べたように、このテストコードにおいて Node クラスの実際の使われ方(ユースケース)を定義している。このテストが成功するように Node クラスの実装を整備していくのだ！

テストコードが通るようになった段階で、Git リポジトリを作成し、コミットしておこう。

```
% git init
% git add .
% git commit -m "initial import"
```

3.2.4. Step3: CrossSectionクラスの拡張

次に、CrossSection クラスをもう少し拡張していこう。最初にやることは･･･そうだ、テストを書くことだ！ test_cross_section.py に次のテストを追加しよう。

test_cross_section.py への追加部分

```
    def test_operation_add(self):
        xs1 = CrossSection([1.0, 2.0, 3.0])
        xs2 = CrossSection([2.0, 3.0, 4.0])
        xs3 = xs1 + xs2
        xs3_ref = CrossSection([3.0, 5.0, 7.0])
        self.assertEqual(xs3, xs3_ref)

    def test_operation_sub(self):
        xs1 = CrossSection([1.0, 2.0, 3.0])
        xs2 = CrossSection([2.0, 3.0, 4.0])
        xs3 = xs2 - xs1
        xs3_ref = CrossSection([1.0, 1.0, 1.0])
        self.assertEqual(xs3, xs3_ref)

    def test_operation_mul(self):
        xs1 = CrossSection([1.0, 2.0, 3.0])
        xs2 = xs1 * 2.0
        xs2_ref = CrossSection([2.0, 4.0, 6.0])
        self.assertEqual(xs2, xs2_ref)

        xs3 = 2.0 * xs1
        xs3_ref = CrossSection([2.0, 4.0, 6.0])
        self.assertEqual(xs3, xs3_ref)

        xs4 = 2.0 * xs1 * 3.0
        xs4_ref = CrossSection([6.0, 12.0, 18.0])
        self.assertEqual(xs4, xs4_ref)
```

先ほどのテストとの違いは次の３つだ。

- コンストラクタに引数を取れるようにした
- 足し算、引き算、掛け算を定義

- オブジェクト同士の比較を定義

以上のコードを追加したテストが成功するように、CrossSectionクラスを拡張していこう。 以下に拡張例を示しておく。

cross_section.py への追加部分

```
    def __eq__(self, other):
        return np.allclose(self.x, other.x)

    def __mul__(self, factor):
        xs = CrossSection(self)
        xs.x *= factor
        return xs

    def __rmul__(self, factor):
        xs = CrossSection(self)
        xs.x *= factor
        return xs

    def __truediv__(self, factor):
        xs = CrossSection(self)
        xs.x *= (1.0/factor)
        return xs

    def __neg__(self):
        return self * (-1.0)

    def __add__(self, other):
        xs = CrossSection(self)
        xs.x += other.x
        return xs

    def __sub__(self, other):
        xs = CrossSection(self) + (-other)
        return xs
```

3.2.5. Step4: Nodeクラスの拡張

次に、Node クラスのテストを拡張しよう。いよいよ、応答行列法を用いた Red/Black Iteration の実装に入ってくる。

べき乗法 (Power Method)による固有値問題の基本的な解法は次のようになる（詳細は、別途理論編を参照されたい）

なお、ここでは zero flux 境界条件を仮定している。基本的なアルゴリズムは次の通りだ。

基本的なアルゴリズム

1. 初期の核分裂中性子源を設定する。体系全体の全核分裂源(total_fis_src) / keff が 1.0となるように規格化する。
2. 内側反復計算において
    - 自ノードへの流入中性子流を準備する （隣のノードの放出中性子）
        - ただし、境界部においては、自分の放出中性子流の符号を反転させたものを用いる
    - 応答行列を用いて、放出中性子流と中性子束を求める
3. 更新された中性子束を用いて核分裂源を計算する
4. 実効増倍率 = 今回の核分裂源 / (前回の核分裂源 / 前回の実効増倍率)

5. 収束していなければ 2. へ戻る

これをテストコードに落とし込んだものが次だ。

ここが本講義で最も重要なステップなので、以下のコードをよーく見て欲しい。

```
import unittest

import sys
sys.path.append('../lib')
import math

from config import *
from cross_section import CrossSection
from node import Node

class NodeTest(unittest.TestCase):

    def test_uniform_zeroflux_bc(self):
        xs_fuel = CrossSection([1.36, 0.0181, 0.0279])
        delta = 1.0
        geom = [{'xs':xs_fuel, 'width':100}]

        nodes = []
        for r in geom:
            for k in range(int(r['width']/delta)):
                the_node = Node(r['xs'])
                the_node.set_width(delta)
                nodes.append(the_node)

        keff = 1.0
        keff_old = 1.0
        total_fis_src_old = 1.0
        conv = 1.0e-7

        for idx_outer in range(2000):  # outer iteration

            # calculation of total fission source
            total_fis_src = 0.0
            for the_node in nodes:
                total_fis_src += the_node.get_fis_src()

            # normalize fis src to make total fis src unity
            norm_factor = 1.0 / (total_fis_src/keff)
            for the_node in nodes:
                the_node.normalize_fis_src(norm_factor)

            # inner iteration with fixed fis src
            for istart in range(2):  # start color (0: red, 1:black)
                for ix in range(istart, len(nodes), 2):
                    if(ix==0):  # left boundary
                        jin_xm = -nodes[ix].get_jout(XM)
                    else:
                        jin_xm = nodes[ix-1].get_jout(XP)

                    if(ix==len(nodes)-1):  # right boundary
                        jin_xp = -nodes[ix].get_jout(XP)
                    else:
                        jin_xp = nodes[ix+1].get_jout(XM)

                    nodes[ix].set_jin(XM, jin_xm)
                    nodes[ix].set_jin(XP, jin_xp)
                    nodes[ix].calc()

            # calculation of new fission source
            for the_node in nodes:
                the_node.calc_fis_src()

            # calculation of total fission source
            total_fis_src = 0.0
            for the_node in nodes:
                total_fis_src += the_node.get_fis_src()

            # estimation of eigen value as a ratio of generations
            keff = total_fis_src / (total_fis_src_old/keff_old)

            diff = abs((keff - keff_old)/keff)
            #print( idx_outer, keff, diff)
```

```
            if(diff < conv):
                break

            keff_old = keff
            total_fis_src_old = total_fis_src

            # set new eigen value to all the nodes
            for the_node in nodes:
                the_node.set_keff(keff)

        # reference as analytical solution
        kana = xs_fuel.nusigf() / (xs_fuel.dif() * math.pi ** 2 / 100**2 +
xs_fuel.siga())
        #print( 'kana = ', kana)
        self.assertAlmostEqual(keff, kana, places=5)
```

いかがだっただろうか？　よく分からなかった？？　基本的にはアルゴリズムを忠実にコードの落とし込んだものだから、理解できるまで何度でもコードを読んで欲しい。

なんとなくでも良いので、理解出来れば次のステップにいくとしよう。

もうお分かりと思うが、このテストコードが成功するように Node クラスを拡張していく。ここで肝となるのが、応答行列法による計算部分だ。詳細は理論編を参照するとして、ここではエッセンスを示しておく。

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。

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

3. 流入中性子流 $J^-_{i\pm1/2}$ とノード平均中止子束 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。

$$J_{i\pm1/2}=\frac{-\frac{2D}{\Delta x}(4J^-_{i\pm1/2}-\phi_i)}{(1+\frac{4D}{\Delta x})}$$

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。

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

上記の解法を calc メソッドに実装し、他のメソッドも準備したものが次のコードだ。

node.py

```
import numpy as np
from cross_section import CrossSection
from config import *

class Node:
    def __init__(self, xs=None):
        self.jout = np.ones(2)    # out-going, [XM, XP]
        self.jin  = np.ones(2)    # in-coming, [XM, XP]
        self.flux  = 1.0    # average flux
```

```
        self.width = 1.0
        self.xs = None
        self.keff = 1.0
        self.fis_src = 1.0
        if(xs):
            self.set_xs(xs)

    def set_xs(self, val):
        self.xs = val

    def set_keff(self, val):
        self.keff = val

    def set_width(self, val):
        self.width = val

    def get_flux(self):
        return self.flux

    def set_jin(self, dir, val):
        self.jin[dir] = val

    def get_jin(self, dir):
        return self.jin[dir]

    def get_jout(self, dir):
        return self.jout[dir]

    def get_xs(self):
        return self.xs

    def get_width(self):
        return self.width

    def get_fis_src(self):
        return self.fis_src

    def calc_fis_src(self):
        # fission source
        self.fis_src = self.xs.nusigf() * self.flux * self.width

    def normalize_fis_src(self, factor):
        self.fis_src *= factor

    def calc(self):
        #flux by Eq(28)
        coef1 = 2.0*self.xs.dif() / self.width
        coef2 = 2.0*coef1
        coef3 = 1.0 + coef2
        f_nume = coef1 * 4.0 * (self.jin[XP] + self.jin[XM]) + \
                 coef3 * self.fis_src / self.keff
        f_deno = coef2 + coef3*self.xs.siga()*self.width
        self.flux = f_nume / f_deno

        #net current by Eq(29)
        jnet_XM  = -coef1 * (4.0*self.jin[XM] - self.flux) / coef3
        jnet_XP = -coef1 * (4.0*self.jin[XP] - self.flux) / coef3

        #out-goinnt by Eq(30)
        self.jout[XM]  = jnet_XM  + self.jin[XM]
        self.jout[XP] = jnet_XP + self.jin[XP]

    def debug(self):
        print("-"*3 + " Node " + "-"*40)
        print("  jin_XM \t", self.jin[XM] )
        print("  jin_XP \t", self.jin[XP])
        print("  jout_XM\t", self.jout[XM])
        print("  jout_XP\t", self.jout[XP])
        print("  flux   \t", self.flux)
        print("  keff   \t", self.keff)
        self.xs.debug()
        print("-"*50)


if __name__ == '__main__':
    node = Node()
    xs = CrossSection()
    xs.set_d(1.0)
    xs.set_siga(2.0)
    xs.set_nusigf(3.0)
    node.set_xs(xs)
    node.debug()
```

3.2.6. Step5: Contaierクラスの新設

さて、テストコードを書くことの3番目のメリットとして下記のものを挙げていたが、覚えているだろうか？

> - テストコードが上位層クラスにおけるメソッドのひな形になる

先ほどの Node クラスに対するテストコードは、計算実行に必要なほぼ完全なコードだと言える。ということは、これを上手くまとめることによって、Nodeクラスを取りまとめる上位層、ここでは Container クラスのメソッドとして再利用することができるのだ。

この時、まずはこんな風にしたいな･･･というイメージを Container クラスのテストコードとして落とし込む。こんな感じだ。

test_container.py

```
import unittest

import sys
sys.path.append('../lib')
import math

from config import *
from cross_section import CrossSection
from node import Node
from container import Container

class ContainerTest(unittest.TestCase):

    def test_container(self):

        xs_fuel = CrossSection([1.36, 0.0181, 0.0279])
        delta = 1.0
        albedo = -1.0
        geom = [{'xs':xs_fuel, 'width':100}]

        cont = Container(geom, delta, albedo)
        #cont.debug()

        keff = 1.0
        keff_old = 1.0
        total_fis_src_old = 1.0
        conv = 1.0e-7

        for idx_outer in range(100):

            total_fis_src = cont.get_total_fis_src()
            norm_factor = 1.0 / (total_fis_src/keff)
            cont.normalize_fis_src(norm_factor)

            for idx_inner in range(4):
                for color in range(2):
                    cont.calc(color)

            cont.calc_fis_src()

            total_fis_src = cont.get_total_fis_src()

            keff = total_fis_src / (total_fis_src_old/keff_old)
            diff = abs((keff - keff_old)/keff)
            #print( keff, diff)
            if(diff < conv):
                break
            keff_old = keff
            total_fis_src_old = total_fis_src

            cont.set_keff(keff)


        kana = xs_fuel.nusigf() / (xs_fuel.dif() * math.pi ** 2 / 100**2 +
xs_fuel.siga())
        #print( 'kana = ', kana)
        self.assertAlmostEqual(keff, kana, places=4)
```

```
        flux = cont.get_flux_dist()
        self.assertEqual(len(flux), 2)  # x, y
        self.assertEqual(len(flux[0]), int(geom[0]['width']/delta))
        self.assertEqual(flux[0][0], delta/2.0)
        self.assertEqual(flux[0][-1], geom[0]['width']-delta/2.0)
        self.assertEqual(len(flux[1]), int(geom[0]['width']/delta))


if __name__ == '__main__':
    unittest.main()
```

コードがかなりすっきりして、全体がすぐに見渡せる程度になってきた。

このテストコードが成功するように Containerクラスを定義するわけだが、その際に先ほどの Nodeクラスのテストコード (test_node.py) を横目にみながら参考にして実装していけば良い。次のような感じだ。

container.py

```
import numpy as np

from node import *
from cross_section import *


class Container:
    """

    """
    def __init__(self, geometry, delta=1.0, albedo=0.0):
        self._setup(geometry, delta, albedo)

    def _setup(self, geometry, delta, albedo):
        self.nodes = []
        for r in geometry:
            for k in range(int(r['width']/delta)):
                the_node = Node(r['xs'])
                the_node.set_width(delta)
                self.nodes.append(the_node)

        self.delta = delta
        self.albedo = albedo

    def calc(self, color=None):
        for ix in range(color, len(self.nodes), 2):
            if(ix==0):
                jin_xm = self.albedo * self.nodes[ix].get_jout(XM)
            else:
                jin_xm = self.nodes[ix-1].get_jout(XP)

            if(ix==len(self.nodes)-1):
                jin_xp = self.albedo * self.nodes[ix].get_jout(XP)
            else:
                jin_xp = self.nodes[ix+1].get_jout(XM)

            self.nodes[ix].set_jin(XM, jin_xm)
            self.nodes[ix].set_jin(XP, jin_xp)
            self.nodes[ix].calc()

    def calc_fis_src(self):
        for the_node in self.nodes:
            the_node.calc_fis_src()

    def get_total_fis_src(self):
        total_fis_src = 0.0
        for the_node in self.nodes:
            total_fis_src += the_node.get_fis_src()
        return total_fis_src

    def normalize_fis_src(self, factor):
        for the_node in self.nodes:
            the_node.normalize_fis_src(factor)

    def set_keff(self, keff):
        for the_node in self.nodes:
```

```
            the_node.set_keff(keff)

    def get_flux_dist(self):
        x_pos = []
        x_sum = 0.0
        flux = []
        for the_node in self.nodes:
            w = the_node.get_width()
            x_pos.append( x_sum + w/2 )
            x_sum += w
            flux.append( the_node.get_flux() )
        return [x_pos, flux]


    def debug(self):
        print("nodes: ", len(self.nodes))
        for the_node in self.nodes:
            the_node.debug()
```

3.2.7. Step6: ContainerControllerクラスの新設

Container クラスのテストコードでもかなりすっきりしていたが、これをもう一段階上のレベルに引き上げてみよう。そう、ContainerController の登場だ。

例のごとく、テストコードを先に書いてみる。

test_container_controller.py

```
import unittest

import sys
sys.path.append('../lib')
import math


from config import *
from cross_section import CrossSection
from node import Node
from container import Container
from container_controller import ContainerController


class ContainerContainerTest(unittest.TestCase):

    def test_container_controller(self):

        xs_fuel = CrossSection([1.36, 0.0181, 0.0279])
        delta = 1.0
        albedo = -1.0
        geom = [{'xs':xs_fuel, 'width':100}]

        container = Container(geom, delta, albedo)
        #cont.debug()

        controller = ContainerController(container)

        controller.calc()

        keff = controller.get_keff()

        kana = xs_fuel.nusigf() / (xs_fuel.dif() * math.pi ** 2 / 100**2 +
xs_fuel.siga())
        #print( 'kana = ', kana)
        self.assertAlmostEqual(keff, kana, places=4)


if __name__ == '__main__':
    unittest.main()
```

ブラボー！　かなりスッキリしてきた！！

いや、まだこれはテストコードだ。このコードが動くように ContainerController 本体を書くんだ！！

書け！　書けっ！　書くんだジョー！

container_controller.py

```
class ContainerController:
    """
    Container
    """

    def __init__(self, container=None):
        self.cont = container
        self.keff = 1.0
        self.keff_old = 1.0
        self.total_fis_src_old = 1.0
        self.conv_criterion = 1.0E-7
        self.max_outer_iterations = 500
        self.inner_iterations = 4
        self.converged = False


    def calc(self):
        for idx_outer in range(self.max_outer_iterations):

            self.total_fis_src = self.cont.get_total_fis_src()
            norm_factor = 1.0 / (self.total_fis_src/self.keff)
            self.cont.normalize_fis_src(norm_factor)

            for idx_inner in range(self.inner_iterations):
                for color in range(2):
                    self.cont.calc(color)

                self.cont.calc_fis_src()

            self.total_fis_src = self.cont.get_total_fis_src()

            self.keff = self.total_fis_src / (self.total_fis_src_old/self.keff_old)
            diff = abs((self.keff - self.keff_old)/self.keff)

            #print(self.keff, diff)

            if(diff < self.conv_criterion):
                self.converged = True
                break

            self.keff_old = self.keff
            self.total_fis_src_old = self.total_fis_src

            self.cont.set_keff(self.keff)

        return (idx_outer, self.converged)


    def get_keff(self):
        return self.keff
```

3.2.8. Step7: CalculationManagerクラスの新設

よし、これでもかなり良いが、もうちょっと抽象化したい。もうちょっとだけだ。

ContainerController のテストコードをもう一段階上のレベルに引き上げよう！ こんな感じだ！

test_calculation_manager.py

```
import unittest

import sys
sys.path.append('../lib')
import math
```

```
from cross_section import CrossSection
from node import Node
from container import Container
from container_controller import ContainerController
from calculation_manager import CalculationManager

class CalculationManagerTest(unittest.TestCase):

    def test_calculation_manager(self):

        xs_fuel = CrossSection([1.36, 0.0181, 0.0279])
        delta = 1.0
        albedo = -1.0
        geom = [{'xs':xs_fuel, 'width':100}]

        config = { 'geometry':geom, 'mesh_width':delta, "albedo": albedo}

        calc_man = CalculationManager(config)
        calc_man.run()

        keff = calc_man.get_keff()

        kana = xs_fuel.nusigf() / (xs_fuel.dif() * math.pi ** 2 / 100**2 +
xs_fuel.siga())
        #print( 'kana = ', kana)
        self.assertAlmostEqual(keff, kana, places=4)


if __name__ == '__main__':
    unittest.main()
```

このレベルまで来ると、計算コードがかなり抽象化されて、取り扱いが非常に楽になる。

ということで、このテストが通るように CalculationManager の本体を実装しよう！

calculation_manager.py

```
from container import Container
from container_controller import ContainerController


class CalculationManager:
    """

    """

    def __init__(self, param):
        geom = param['geometry']
        delta = param['mesh_width']
        albedo = param['albedo']

        container = Container(geom, delta, albedo)
        self.controller = ContainerController(container)

    def run(self):
        self.controller.calc()

    def get_keff(self):
        return self.controller.get_keff()
```

今度は結構短いな･･･。これで一通りのコーディングは完了した。

テストが通ることを確認したら、リポジトリにコミットしておこう。

3.2.9. Final: 完成

これでいよいよ完成だ。最終的にできあがった計算コードは次のようになる。

bare_1g.py

```
import sys

sys.path.append('lib')
from calculation_manager import CalculationManager
from cross_section import CrossSection

xs_fuel = CrossSection([1.36, 0.0181, 0.0279])
delta = 1.0
albedo = -1.0
geom = [{'xs':xs_fuel, 'width':100}]

config = { 'geometry':geom, 'mesh_width':delta, "albedo": albedo}

calc_man = CalculationManager(config)
calc_man.run()

keff = calc_man.get_keff()

print ("keff = ", keff)
```

もう気づいたと思うが、実はこれは CalculatiomNanager のテストコードそのものだ。

今まで書いてきた部分は、いわゆる計算カーネル (claculation kernel) と言うべき中心的部分だ。実際の計算コードでのでは、計算カーネルはむしろ20%程度で、入出力とそれに関わるエラーチェックが80%程度を占めるといっても過言では無い。本講義での目的は、プログラミングの方法論の根幹部分について議論することである。したがって、入出力等の補助的な部分については割愛するが、ニーズに応じて読者の自学自習に期待したい。

ここまでにおいて、CrossSection クラスや Node クラスといった基本的なクラスの定義やテストコードの作成から始まり、徐々に上位の階層におけるテストコードの作成を通じて、最終目的である計算コードの作成にたどり着くことが出来た。

もちろん、必ずしも常にボトムアップでスムーズに開発出来るとは限らない。上位層の設計・実装が思惑通り行かず、階層の設計・実装をやり直すということは良くあることである。多かれ少なかれ試行錯誤は発生するだろうが、経験をつんで自分なりの定石のパターンを築き上げれば、手戻りは少なくなって行くであろう。

また、理論編でも述べたように、デザインパターン、アルゴリズム、実装テクニックについては、書籍や他のソースコードをたくさん読むことにより効果的に学ぶことが出来るので、是非ともチャレンジして欲しい。

3.2.10. 演習問題

1. 裸の原子炉における中性子束分布 (零中性子束境界条件)

完成した計算コードで求めた中性子束分布をプロットさせてみよう。中性子束分布は、Container オブジェクトにおける get_flux_dist メソッドで下記のように取得できるようにしたい。

```
flux_dist = calc_man.controller.container.get_flux_dist()
```

戻り値として、下記のような2次元リストを返すように設定すること。

```
[[0.5,
  1.5,
```

```
   ...
   98.5,
   99.5],
  [0.012688756616239107,
   0.038053732231784085,
   ...
   0.03806177262712973,
   0.012691395758198915]]
```

下記のコードを用いて、得られた中性子束を Jupyter Notebook で可視化してみよう。

```
%matplotlib notebook
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(flux_dist[0], flux_dist[1])
plt.show()
```

2. 反射体付き原子炉における実効増倍率と中性子束分布

厚さ60cmの燃料の両端に厚さ30cmの軽水反射体が配置されている1次元平板原子炉を考えて、その実効増倍率と中性子束分布を可視化してみよう。なお、断面積は下記のものを用いること。

```
xs_fuel = CrossSection([1.36, 0.0181, 0.0279])
xs_ref  = CrossSection([0.55, 0.0127, 0.0])
```

3.3. 2群計算への拡張

前節では1群計算を対象としていたが、これを2群計算に拡張してみよう。以下に1群計算と2群計算の違いをまとめる。

- 断面積について
  - エネルギー群数が2倍 (引数kgの追加)
  - 核分裂スペクトルの取り扱い
  - 散乱マトリクスの取り扱い
- 収束計算
  - エネルギー群による反復計算
  - 散乱中性子源の計算

これらの違いを意識しながら、下位層のクラスから少しずつ変更していこう。

3.3.1. CrossSectionクラス

まずは、CrossSection クラスのテストコードを2群計算に対応すべく変更しよう。

定数設定のためのメソッドについては、第一引数としてエネルギー群を指定できるようにする。また、核分裂スペクトルの設定 (set_xi) も追加しよう。さらに、散乱マトリクス (scattering matrix) を設定するメソッド(set_smat) を追加しよう。また、設定した値を取得できる getter メソッドも必要だ。

以上を考慮して作成したテストコードが次のものだ。

test_cross_section.py

```
import unittest

import sys
sys.path.append('../lib')

from node import Node
from cross_section import CrossSection

class NodeTest(unittest.TestCase):

    def test_sets(self):
        xs = CrossSection()
        xs.set_d(0, 1.0)
        xs.set_siga(0, 2.0)
        xs.set_nusigf(0, 3.0)
        xs.set_xi(0, 1.0)
        xs.set_d(1, 11.0)
        xs.set_siga(1, 12.0)
        xs.set_nusigf(1, 13.0)
        xs.set_xi(1, 0.0)
        xs.set_smat([[1.0, 2.0],[3.0, 4.0]])
        #   sm(kg, kkg)
        #
        #             kkg
        #           0     1
        #   kg 0   1.0   2.0
        #      1   3.0   4.0

        self.assertEqual(xs.dif(0), 1.0)
        self.assertEqual(xs.siga(0), 2.0)
        self.assertEqual(xs.nusigf(0), 3.0)
        self.assertEqual(xs.xi(0), 1.0)
        self.assertEqual(xs.dif(1), 11.0)
        self.assertEqual(xs.siga(1), 12.0)
        self.assertEqual(xs.nusigf(1), 13.0)
        self.assertEqual(xs.xi(1), 0.0)
        self.assertEqual(xs.sigs(0,0), 1.0)
        self.assertEqual(xs.sigs(0,1), 2.0)
        self.assertEqual(xs.sigs(1,0), 3.0)
        self.assertEqual(xs.sigs(1,1), 4.0)

    def test_sets3(self):
        xs1 = CrossSection()
        xs1.set([[1.0, 2.0, 3.0, 1.0],[11.0, 12.0, 13.0, 0.0]])
        xs1.set_smat( [[1.0, 2.0], [3.0, 4.0]])
        xs1_ref = CrossSection()
        xs1_ref.set(xs1)
        self.assertEqual(xs1, xs1_ref)

    def test_operation_add(self):
        xs1 = CrossSection()
        xs1.set([[1.0, 2.0, 3.0, 1.0],[11.0, 12.0, 13.0, 0.0]])
        xs1.set_smat( [[1.0, 2.0], [3.0, 4.0]])

        xs2 = CrossSection()
        xs2.set([[1.0, 2.0, 3.0, 1.0],[11.0, 12.0, 13.0, 0.0]])
        xs2.set_smat( [[1.0, 2.0], [3.0, 4.0]])

        xs3 = xs1 + xs2

        xs3_ref = CrossSection()
        xs3_ref.set([[2.0, 4.0, 6.0, 2.0],[22.0, 24.0, 26.0, 0.0]])
        xs3_ref.set_smat( [[2.0, 4.0], [6.0, 8.0]])

        self.assertEqual(xs3, xs3_ref)

    def test_operation_sub(self):
        xs1 = CrossSection()
        xs1.set([[1.0, 2.0, 3.0, 1.0],[11.0, 12.0, 13.0, 0.0]])
        xs1.set_smat( [[1.0, 2.0], [3.0, 4.0]])

        xs2 = CrossSection()
        xs2.set([[2.0, 4.0, 6.0, 2.0],[22.0, 24.0, 26.0, 0.0]])
        xs2.set_smat( [[2.0, 4.0], [6.0, 8.0]])

        xs3 = xs2 - xs1

        xs3_ref = CrossSection()
        xs3_ref.set([[1.0, 2.0, 3.0, 1.0],[11.0, 12.0, 13.0, 0.0]])
        xs3_ref.set_smat( [[1.0, 2.0], [3.0, 4.0]])

        self.assertEqual(xs3, xs3_ref)
```

```
    def test_operation_mul(self):
        xs1 = CrossSection()
        xs1.set([[1.0, 2.0, 3.0, 1.0],[11.0, 12.0, 13.0, 0.0]])
        xs1.set_smat( [[1.0, 2.0], [3.0, 4.0]])

        xs2 = xs1 * 2.0

        xs2_ref = CrossSection()
        xs2_ref.set([[2.0, 4.0, 6.0, 2.0],[22.0, 24.0, 26.0, 0.0]])
        xs2_ref.set_smat( [[2.0, 4.0], [6.0, 8.0]])

        self.assertEqual(xs2, xs2_ref)

        xs3 = 2.0 * xs1
        xs3_ref = xs2_ref
        self.assertEqual(xs3, xs3_ref)

        xs4 = 2.0 * xs1 * 3.0
        xs4_ref = CrossSection()
        xs4_ref.set([[6.0, 12.0, 18.0, 6.0],[66.0, 72.0, 78.0, 0.0]])
        xs4_ref.set_smat( [[6.0, 12.0], [18.0, 24.0]])
        xs4.debug()
        xs4_ref.debug()

        self.assertEqual(xs4, xs4_ref)


if __name__ == '__main__':
    unittest.main()
```

そして、このテストコードが成功するように CrossSection クラスを実装していく。以下に例を示しておこう。

cross_section.py

```
import numpy as np
import copy

#
N_REACT = 5  # D, Siga, nuSigf, Xi, sigr
DIF    = 0
SIGA   = 1
NUSIGF = 2
XI     = 3
SIGR   = 4

class CrossSection:
    """
            (2  )
    """

    def __init__(self, val=None, ng=2):
        """

        """
        self.x  = np.zeros((ng, N_REACT))
        self.sm = np.zeros((ng, ng))
        if not (val is None):
            self.set(val)

    def set(self, val):
        if type(val) == CrossSection:
            self.x = copy.copy(val.x)
            self.sm = copy.copy(val.sm)
        else: #
            for kg in range(self.ng()):
                for i in range(len(val[kg])):
                    self.x[kg, i] = val[kg][i]

    def set_d(self, kg, val):
        self.x[kg, DIF] = val

    def set_siga(self, kg, val):
        self.x[kg, SIGA] = val

    def set_nusigf(self, kg, val):
```

```
        self.x[kg, NUSIGF] = val

    def set_xi(self, kg, val):
        self.x[kg, XI] = val

    def set_smat(self, mat):
        for kg in range(self.ng()):
            for i in range(self.ng()):
                self.sm[kg, i] = mat[kg][i]

    def calc_sigr(self):
        for kg in range(self.ng()):
            self.x[kg, SIGR] = self.x[kg, SIGA]
            for kkg in range(self.ng()):
                if kg != kkg:
                    self.x[kg, SIGR] += self.sm[kg,kkg]

    def ng(self):
        return self.x.shape[0]

    def dif(self, kg):
        return self.x[kg, DIF]

    def siga(self, kg):
        return self.x[kg, SIGA]

    def sigr(self, kg):
        return self.x[kg, SIGR]

    def nusigf(self, kg):
        return self.x[kg, NUSIGF]

    def xi(self, kg):
        return self.x[kg, XI]

    def sigs(self, kg, kkg):
        return self.sm[kg, kkg]

    def __eq__(self, other):
        return np.allclose(self.x, other.x) and np.allclose(self.sm, other.sm)

    def __mul__(self, factor):
        #       val    float
        xs = CrossSection(self)
        xs.x *= factor
        xs.sm *= factor
        return xs

    def __rmul__(self, factor):
        xs = CrossSection(self)
        xs.x *= factor
        xs.sm *= factor
        return xs

    def __truediv__(self, factor):
        xs = CrossSection(self)
        xs.x *= (1.0/factor)
        xs.sm *= (1.0/factor)
        return xs

    def __neg__(self):
        return self * (-1.0)

    def __add__(self, other):
        xs = CrossSection(self)
        xs.x += other.x
        xs.sm += other.sm
        return xs

    def __sub__(self, other):
        xs = CrossSection(self) + (-other)
        return xs

    def debug(self):
        print("-" * 9 + " XS " + "-" * 29)
        print("kg\tD\tSiga\tSigr\tNuSigf\tXi")
        for kg in range(self.x.shape[0]):
            print(kg, self.x[kg, DIF], self.x[kg, SIGA], self.x[kg, SIGR], self.x[kg,
NUSIGF], self.x[kg, XI], sep='\t', end='\n')
        print( "smat")
        print( self.sm )
        print("-"*42)
```

3.3.2. Nodeクラス

次に Node クラス周りをみていこう。例のごとく、テストコードから。

2群計算になって大きく変わるところは、エネルギーループが必要なことと、散乱ソースの取り扱いだ。

問題を可能な限りシンプルにするために、最初のテストではノードの数を一つとした。これは、2群計算の導入による違いを主としてテストすることができるからだ。このように、「違いをもたらす違い (The difference makes the difference)」に集中した「最小問題セット (the minimum problem set)」のテストを書いて、そのテストを通すように本体を実装していくのが重要だ。初期のテストが通った段階で、必要に応じてテストの数を増やしていけば良い。

少し長くなるが、このような方法で作成したテストコードとクラス実装の例を以下に示しておく。

test_node.py

```
import unittest

import sys
sys.path.append('../lib')
import math

from config import *
from cross_section import CrossSection
from node import Node

class NodeTest(unittest.TestCase):

    def test_onenode(self):
        node = Node()
        xs = CrossSection()

        # two-group problem
        xs.set([[1.58, 0.0032, 0.0, 1.0],[0.271, 0.0930, 0.168, 0.0]])
        xs.set_smat( [[0.0, 0.0178], [0.0, 0.0]])
        xs.calc_sigr()

        #xs.debug()
        node.set_xs(xs)

        keff = 1.0
        keff_old = 1.0
        total_fis_src_old = 1.0
        conv = 1.0e-10

        # outer iteration
        for ik in range(1000):

            # normalize total fission source
            total_fis_src = 0.0
            for kg in range(2):
                total_fis_src += node.get_fis_src(kg) * node.get_width()
            factor = 1.0 / (total_fis_src/keff)

            for kg in range(2):
                node.normalize_fis_src(kg, factor)

            # energy loop
            for kg in range(2):
                # update sources
                node.calc_scat_src(kg)

                # inner loop
                for i in range(4):
                    for dir in range(2):
                        node.set_jin(kg, dir, node.get_jout(kg, dir))

                    # calculate jout, flux with response matrix
                    node.calc(kg)

                node.calc_fis_src(kg)

            # calc total fission source and k_eff
```

```
            total_fis_src = 0.0
            for kg in range(2):
                total_fis_src += node.get_fis_src(kg) * node.get_width()

            keff = total_fis_src / (total_fis_src_old/keff_old)
            diff = abs((keff - keff_old)/keff)

            # convergence check
            if(diff < conv):
                break

            # update parameters
            total_fis_src_old = total_fis_src
            keff_old = keff
            node.set_keff(keff)

        # --- end of loop for outer iteration

        print("keff=", keff)

        kana_nume = xs.sigr(1)*xs.nusigf(0) + xs.sigs(0,1)*xs.nusigf(1)
        kana_deno = xs.sigr(0) * xs.sigr(1)
        kana = kana_nume / kana_deno
        print("kana=", kana)

        self.assertAlmostEqual(keff, kana, places=5)

        for kg in range(2):
            self.assertAlmostEqual(node.get_jout(kg, XM), node.get_jout(kg, XP),
places=5)
            self.assertAlmostEqual(node.get_jin(kg, XM), node.get_jout(kg, XM),
places=5)
            self.assertAlmostEqual(node.get_jin(kg, XP), node.get_jout(kg, XP),
places=5)
            self.assertAlmostEqual(node.get_jout(kg, XM)+node.get_jin(kg, XM),
node.get_flux(kg) / 2.0, places=5)


    def test_uniform_zeroflux_bc(self):
        xs = CrossSection()
        xs.set([[1.58, 0.02, 0.0, 1.0],[0.271, 0.0930, 0.168, 0.0]])
        xs.set_smat( [[0.0, 0.0178], [0.0, 0.0]])
        xs.calc_sigr()

        delta = 1.0
        geom = [{'xs':xs, 'width':100}]

        # geometry setting
        nodes = []
        for r in geom:
            for k in range(int(r['width']/delta)):
                the_node = Node(r['xs'])
                the_node.set_width(delta)
                nodes.append(the_node)

        keff = 1.0
        keff_old = 1.0
        total_fis_src_old = 1.0
        conv = 1.0e-8

        # outer iteration
        for ik in range(100):

            # normalize total fission source
            total_fis_src = 0.0
            for the_node in nodes:
                for kg in range(2):
                    total_fis_src += the_node.get_fis_src(kg) * the_node.get_width()
            factor = 1.0 / (total_fis_src/keff)

            for the_node in nodes:
                for kg in range(2):
                    the_node.normalize_fis_src(kg, factor)

            # energy loop
            for kg in range(2):

                # update scattering source
                for the_node in nodes:
                    the_node.calc_scat_src(kg)

                # inner loop
                for i in range(4):
                    for istart in range(2):  # start color (0: red, 1:black)
```

```
                        for ix in range(istart, len(nodes), 2):

                            # pass partial currents to adjacent nodes
                            if(ix==0):
                                jin_xm = -nodes[ix].get_jout(kg, XM)
                            else:
                                jin_xm = nodes[ix-1].get_jout(kg, XP)

                            if(ix==len(nodes)-1):
                                jin_xp = -nodes[ix].get_jout(kg, XP)
                            else:
                                jin_xp = nodes[ix+1].get_jout(kg, XM)

                            nodes[ix].set_jin(kg, XM, jin_xm)
                            nodes[ix].set_jin(kg, XP, jin_xp)

                            # calculate jout, flux with response matrix
                            nodes[ix].calc(kg)

                    # update fission source
                    for the_node in nodes:
                        the_node.calc_fis_src(kg)

                # calc total fission source and keff
                total_fis_src = 0.0
                for the_node in nodes:
                    for kg in range(2):
                        total_fis_src += the_node.get_fis_src(kg) * the_node.get_width()

                keff = total_fis_src / (total_fis_src_old/keff_old)
                diff = abs((keff - keff_old)/keff)
                # print( keff, diff)

                # convergence check
                if(diff < conv):
                    break

                # update parameters
                total_fis_src_old = total_fis_src
                keff_old = keff
                for the_node in nodes:
                    the_node.set_keff(keff)

            # --- end of loop for outer iterations

            # converged
            print("keff=", keff)

            # debug
            #print("fast flux")
            #for ix in range(len(nodes)):
            #    print(ix, nodes[ix].get_flux(0))

            b2 = (math.pi / geom[0]['width'])**2

            kana_nume = (xs.sigr(1) + xs.dif(1)*b2)*xs.nusigf(0) +
xs.sigs(0,1)*xs.nusigf(1)
            kana_deno = (xs.dif(0)*b2 + xs.sigr(0) ) * (xs.dif(1)*b2 + xs.sigr(1))
            kana = kana_nume / kana_deno
            print("kana=", kana)

            self.assertAlmostEqual(keff, kana, places=4)



        def test_two_regions_zeroflux_bc(self):

            xs_fuel = CrossSection()
            xs_fuel.set([[1.58, 0.0032, 0.0, 1.0],[0.271, 0.0930, 0.168, 0.0]])
            xs_fuel.set_smat( [[0.0, 0.0178], [0.0, 0.0]])
            xs_fuel.calc_sigr()

            xs_ref = CrossSection()
            xs_ref.set([[1.41, 0.0, 0.0, 1.0],[0.117, 0.0191, 0.0, 0.0]])
            xs_ref.set_smat( [[0.0, 0.0476], [0.0, 0.0]])
            xs_ref.calc_sigr()

            delta = 1.0
            geom = [{'xs':xs_ref, 'width':30}, {'xs':xs_fuel, 'width':60}, {'xs':xs_ref,
'width':30} ]

            # geometry setting
            nodes = []
            for r in geom:
```

```
            for k in range(int(r['width']/delta)):
                the_node = Node(r['xs'])
                the_node.set_width(delta)
                nodes.append(the_node)

        keff = 1.0
        keff_old = 1.0
        total_fis_src_old = 1.0
        conv = 1.0e-8

        # outer iteration
        for ik in range(100):

            # normalize total fission source
            total_fis_src = 0.0
            for the_node in nodes:
                for kg in range(2):
                    total_fis_src += the_node.get_fis_src(kg) * the_node.get_width()
            factor = 1.0 / (total_fis_src/keff)

            for the_node in nodes:
                for kg in range(2):
                    the_node.normalize_fis_src(kg, factor)

            # energy loop
            for kg in range(2):

                # update scattering source
                for the_node in nodes:
                    the_node.calc_scat_src(kg)

                # inner loop
                for i in range(4):
                    for istart in range(2):  # start color (0: red, 1:black)
                        for ix in range(istart, len(nodes), 2):

                            # pass partial currents to adjacent nodes
                            if(ix==0):
                                jin_xm = -nodes[ix].get_jout(kg, XM)
                            else:
                                jin_xm = nodes[ix-1].get_jout(kg, XP)

                            if(ix==len(nodes)-1):
                                jin_xp = -nodes[ix].get_jout(kg, XP)
                            else:
                                jin_xp = nodes[ix+1].get_jout(kg, XM)

                            nodes[ix].set_jin(kg, XM, jin_xm)
                            nodes[ix].set_jin(kg, XP, jin_xp)

                            # calculate jout, flux with response matrix
                            nodes[ix].calc(kg)

                # update fission source
                for the_node in nodes:
                    the_node.calc_fis_src(kg)

            # calc total fission source and keff
            total_fis_src = 0.0
            for the_node in nodes:
                for kg in range(2):
                    total_fis_src += the_node.get_fis_src(kg) * the_node.get_width()

            keff = total_fis_src / (total_fis_src_old/keff_old)
            diff = abs((keff - keff_old)/keff)

            # print( keff, diff)
            # convergence check
            if(diff < conv):
                break

            # update parameters
            total_fis_src_old = total_fis_src
            keff_old = keff
            for the_node in nodes:
                the_node.set_keff(keff)

        # --- end of loop for outer iterations

        # converged
        print("keff=", keff)

        #print("flux")
        #for ix in range(len(nodes)):
```

```
        #   print(ix, nodes[ix].get_flux())

        self.assertAlmostEqual(keff, 1.35826, places=5)  # keff with strict condition

if __name__ == '__main__':
    unittest.main()
```

node.py

```
import numpy as np

from cross_section import *
from config import *

class Node:
    def __init__(self, xs=None, kg=2):
        self.jout = np.ones((kg,2))   # kg, out-going, [XM, XP]
        self.jin  = np.ones((kg,2))   # kg, in-coming, [XM, XP]
        self.flux  = np.ones(kg)      # average flux
        self.width = 1.0
        self.xs = None
        self.keff = 1.0
        self.fis_src = np.ones(kg)
        self.scat_src = np.zeros(kg)
        if(xs):
            self.set_xs(xs)

    def set_xs(self, val):
        self.xs = val

    def set_keff(self, val):
        self.keff = val

    def set_width(self, val):
        self.width = val

    def set_flux(self, kg, val):
        self.flux[kg] = val

    def get_flux(self, kg):
        return self.flux[kg]

    def set_jin(self, kg, dir, val):
        self.jin[kg, dir] = val

    def get_jin(self, kg, dir):
        return self.jin[kg, dir]

    def get_jout(self, kg, dir):
        return self.jout[kg, dir]

    def get_xs(self):
        return self.xs

    def get_width(self):
        return self.width

    def get_fis_src(self, kg):
        return self.fis_src[kg]

    def calc_fis_src(self, kg):
        # fission source
        s_fis = 0.0
        for kkg in range(self.xs.ng()):
            s_fis += self.xs.xi(kg) * self.xs.nusigf(kkg)*self.flux[kkg]
        self.fis_src[kg] = s_fis

    def calc_scat_src(self, kg):
        # scattering source
        s_scat = 0.0
        for kkg in range(self.xs.ng()):
            if kg != kkg:
                s_scat += self.xs.sigs(kkg, kg) * self.flux[kkg]
        self.scat_src[kg] = s_scat

    def normalize_fis_src(self, kg, factor):
        self.fis_src[kg] *= factor

    def calc(self, kg):
```

```
        #flux by Eq(28)
        coef1 = 2.0*self.xs.dif(kg) / self.width
        coef2 = 2.0*coef1
        coef3 = 1.0 + coef2
        f_nume = coef1 * 4.0 * (self.jin[kg, XP] + self.jin[kg, XM]) + coef3 * \
               (self.fis_src[kg]  / self.keff + self.scat_src[kg]) * self.width
        f_deno = coef2 + coef3*self.xs.sigr(kg)*self.width
        self.flux[kg] = f_nume / f_deno

        #net current by Eq(29)
        jnet_XM  = -coef1 * (4.0*self.jin[kg, XM] - self.flux[kg]) / coef3
        jnet_XP = -coef1 * (4.0*self.jin[kg, XP] - self.flux[kg]) / coef3

        #out-goinnt by Eq(30)
        self.jout[kg, XM] = jnet_XM + self.jin[kg, XM]
        self.jout[kg, XP] = jnet_XP + self.jin[kg, XP]


    def debug(self):
        print("-"*3 + " Node " + "-"*40)
        print( "kg\tjin_XM\tjin_XP\tjout_XM\tjout_XP")
        for kg in range(self.xs.ng()):
            print(kg, self.jin[kg, XM], self.jin[kg, XP], self.jout[kg, XM],
self.jout[kg, XP], sep='\t')

        print("  flux   \t", self.flux)
        print("  fis_src\t", self.fis_src)
        print("  scat_src\t", self.scat_src)
        print("  keff   \t", self.keff)

        self.xs.debug()
        print("-"*50)
```

3.3.3. Containerクラス

Container クラスは内部反復を担当するとした。すなわち、calc メソッドの引数にはエネルギー群である kg を渡して、エネルギー反復を Container クラスの外側で設定する。2群化することで、散乱中性子源の計算部分も追加している。以下に、Container クラスのテストコードにそのロジックの例を示す。

test_container.py

```
import unittest

import sys
sys.path.append('../lib')
import math

from config import *
from cross_section import CrossSection
from node import Node
from container import Container

class ContainerTest(unittest.TestCase):

    def test_container(self):

        xs = CrossSection()
        xs.set([[1.58, 0.02, 0.0, 1.0],[0.271, 0.0930, 0.168, 0.0]])
        xs.set_smat( [[0.0, 0.0178], [0.0, 0.0]])
        xs.calc_sigr()

        delta = 1.0
        albedo = -1.0
        geom = [{'xs':xs, 'width':100}]

        cont = Container(geom, delta, albedo)
        #cont.debug()

        keff = 1.0
        keff_old = 1.0
        total_fis_src_old = 1.0
        conv = 1.0e-7

        for idx_outer in range(100):
```

```
            total_fis_src = cont.get_total_fis_src()
            norm_factor = 1.0 / (total_fis_src/keff)
            cont.normalize_fis_src(norm_factor)

            for kg in range(2):
                cont.calc_scat_src(kg)

                for idx_inner in range(4):
                    for color in range(2):
                        cont.calc(kg, color)

                cont.calc_fis_src(kg)

            total_fis_src = cont.get_total_fis_src()

            keff = total_fis_src / (total_fis_src_old/keff_old)
            diff = abs((keff - keff_old)/keff)
            #print( keff, diff)
            if(diff < conv):
                break
            keff_old = keff
            total_fis_src_old = total_fis_src

            cont.set_keff(keff)

        b2 = (math.pi / geom[0]['width'])**2
        kana_nume = (xs.sigr(1) + xs.dif(1)*b2)*xs.nusigf(0) +
xs.sigs(0,1)*xs.nusigf(1)
        kana_deno = (xs.dif(0)*b2 + xs.sigr(0) ) * (xs.dif(1)*b2 + xs.sigr(1))
        kana = kana_nume / kana_deno
        print("kana=", kana)

        self.assertAlmostEqual(keff, kana, places=4)


        flux = cont.get_flux_dist(0)  # first energy
        self.assertEqual(len(flux), 2)  # x, y
        self.assertEqual(len(flux[0]), int(geom[0]['width']/delta))
        self.assertEqual(flux[0][0], delta/2.0)
        self.assertEqual(flux[0][-1], geom[0]['width']-delta/2.0)
        self.assertEqual(len(flux[1]), int(geom[0]['width']/delta))


if __name__ == '__main__':
    unittest.main()
```

毎度のことだが、このテストが通るように Container クラスを実装していく。

container.py

```
import numpy as np

from node import *
from cross_section import *


class Container:
    def __init__(self, geometry, delta=1.0, albedo=0.0):
        self._setup(geometry, delta, albedo)

    def _setup(self, geometry, delta, albedo):
        self.nodes = []
        for r in geometry:
            for k in range(int(r['width']/delta)):
                the_node = Node(r['xs'])
                the_node.set_width(delta)
                self.nodes.append(the_node)

        self.delta = delta
        self.albedo = albedo
        self.ng = self.nodes[0].get_xs().ng()

    def get_ng(self):
        return self.ng


    def calc(self, kg, color=None):
        for ix in range(color, len(self.nodes), 2):
```

```
            if(ix==0):
                jin_xm = self.albedo * self.nodes[ix].get_jout(kg, XM)
            else:
                jin_xm = self.nodes[ix-1].get_jout(kg, XP)

            if(ix==len(self.nodes)-1):
                jin_xp = self.albedo * self.nodes[ix].get_jout(kg, XP)
            else:
                jin_xp = self.nodes[ix+1].get_jout(kg, XM)

            self.nodes[ix].set_jin(kg, XM, jin_xm)
            self.nodes[ix].set_jin(kg, XP, jin_xp)
            self.nodes[ix].calc(kg)

    def calc_fis_src(self, kg):
        for the_node in self.nodes:
            the_node.calc_fis_src(kg)

    def calc_scat_src(self, kg):
        for the_node in self.nodes:
            the_node.calc_scat_src(kg)

    def get_total_fis_src(self):
        total_fis_src = 0.0
        for kg in range(self.ng):
            for the_node in self.nodes:
                total_fis_src += the_node.get_fis_src(kg) * the_node.get_width()
        return total_fis_src

    def normalize_fis_src(self, factor):
        for kg in range(self.ng):
            for the_node in self.nodes:
                the_node.normalize_fis_src(kg, factor)

    def set_keff(self, keff):
        for the_node in self.nodes:
            the_node.set_keff(keff)

    def get_flux_dist(self, kg):
        x_pos = []
        x_sum = 0.0
        flux = []
        for the_node in self.nodes:
            w = the_node.get_width()
            x_pos.append( x_sum + w/2 )
            x_sum += w
            flux.append( the_node.get_flux(kg) )
        return [x_pos, flux]


    def debug(self):
        print("nodes: ", len(self.nodes))
        for the_node in self.nodes:
            the_node.debug()
```

3.3.4. ContainerController クラス

ContainerController のレベルになれば、テストコードで準備する断面積や解析解の計算式以外に、1群計算と2群計算の間で実質的な差異はない。

test_container_controller.py

```
import unittest

import sys
sys.path.append('../lib')
import math


from config import *
from cross_section import CrossSection
from node import Node
from container import Container
from container_controller import ContainerController
```

```
class ContainerContainerTest(unittest.TestCase):

    def test_container_controller(self):
        xs = CrossSection()
        xs.set([[1.58, 0.02, 0.0, 1.0],[0.271, 0.0930, 0.168, 0.0]])
        xs.set_smat( [[0.0, 0.0178], [0.0, 0.0]])
        xs.calc_sigr()

        delta = 1.0
        albedo = -1.0
        geom = [{'xs':xs, 'width':100}]

        container = Container(geom, delta, albedo)
        #cont.debug()

        controller = ContainerController(container)

        count, flag = controller.calc()
        print("outer iterations: ", count)

        keff = controller.get_keff()

        b2 = (math.pi / geom[0]['width'])**2
        kana_nume = (xs.sigr(1) + xs.dif(1)*b2)*xs.nusigf(0) +
xs.sigs(0,1)*xs.nusigf(1)
        kana_deno = (xs.dif(0)*b2 + xs.sigr(0) ) * (xs.dif(1)*b2 + xs.sigr(1))
        kana = kana_nume / kana_deno
        print("kana=", kana)

        self.assertAlmostEqual(keff, kana, places=4)


if __name__ == '__main__':
    unittest.main()
```

ContainerController の実装については、Containerクラスのテストコードでの修正がそのまま取り込まれた形となっている。すなわち、エネルギーループの追加と、散乱ソースの計算部分の追加が行われている。

container_controller.py

```
class ContainerController:
    """
    Container
    """

    def __init__(self, container=None):
        self.cont = container
        self.keff = 1.0
        self.keff_old = 1.0
        self.total_fis_src_old = 1.0
        self.conv_criterion = 1.0E-7
        self.max_outer_iterations = 500
        self.inner_iterations = 4
        self.converged = False

    def calc(self):
        for idx_outer in range(self.max_outer_iterations):

            self.total_fis_src = self.cont.get_total_fis_src()
            norm_factor = 1.0 / (self.total_fis_src/self.keff)
            self.cont.normalize_fis_src(norm_factor)

            for kg in range(self.cont.get_ng()):
                self.cont.calc_scat_src(kg)

                for idx_inner in range(self.inner_iterations):
                    for color in range(2):
                        self.cont.calc(kg, color)

                    self.cont.calc_fis_src(kg)

                if self.cont.is_asymptotic(kg):
                    self.cont.accel_flux(kg)
                self.cont.save_old_flux(kg)
```

```
            self.total_fis_src = self.cont.get_total_fis_src()

            self.keff = self.total_fis_src / (self.total_fis_src_old/self.keff_old)
            diff = abs((self.keff - self.keff_old)/self.keff)

            #print(self.keff, diff)

            if(diff < self.conv_criterion):
                self.converged = True
                break

            self.keff_old = self.keff
            self.total_fis_src_old = self.total_fis_src

            self.cont.set_keff(self.keff)

        return (idx_outer, self.converged)

    def get_keff(self):
        return self.keff
```

3.3.5. CalculationManager

CalculationManager についても殆ど差異はないが、参考のため、テストコードと本体の実装例を示しておく。

test_calculation_manager.y

```
import unittest

import sys
sys.path.append('../lib')
import math

from config import *
from cross_section import CrossSection
from node import Node
from container import Container
from container_controller import ContainerController
from calculation_manager import CalculationManager

class CalculationManagerTest(unittest.TestCase):

    def test_calculation_manager(self):

        xs = CrossSection()
        xs.set([[1.58, 0.02, 0.0, 1.0],[0.271, 0.0930, 0.168, 0.0]])
        xs.set_smat( [[0.0, 0.0178], [0.0, 0.0]])
        xs.calc_sigr()

        delta = 1.0
        albedo = -1.0
        geom = [{'xs':xs, 'width':100}]

        config = { 'geometry':geom, 'mesh_width':delta, "albedo": albedo}

        calc_man = CalculationManager(config)
        count, flag = calc_man.run()

        keff = calc_man.get_keff()

        b2 = (math.pi / geom[0]['width'])**2
        kana_nume = (xs.sigr(1) + xs.dif(1)*b2)*xs.nusigf(0) +
xs.sigs(0,1)*xs.nusigf(1)
        kana_deno = (xs.dif(0)*b2 + xs.sigr(0) ) * (xs.dif(1)*b2 + xs.sigr(1))
        kana = kana_nume / kana_deno
        print("kana=", kana)

        self.assertAlmostEqual(keff, kana, places=4)


if __name__ == '__main__':
    unittest.main()
```

calculation_manager.py

```
from container import Container
from container_controller import ContainerController

class CalculationManager:
    """

    """
    def __init__(self, param):
        geom = param['geometry']
        delta = param['mesh_width']
        albedo = param['albedo']

        container = Container(geom, delta, albedo)
        self.controller = ContainerController(container)

    def run(self):
        return self.controller.calc()

    def get_keff(self):
        return self.controller.get_keff()
```

3.4. 加速法の実装

さて、2群計算コードが出来たわけだが、まだまだプロトタイプの段階にすぎず、実用にはほど遠い。というのも、計算アルゴリズムが単純な Red/Black法とべき乗法 (Power Method）を組み合わせたものだからだ。これは何を意味するのか？　いろんな計算を試してみると分かるのだが、計算が非常に「重い」のである。

1次元体系での2群計算ならそれほどでも無いが、3次元の詳細メッシュ体系で多群計算をしようものなら目も当てられない程に、収束がとても遅いのである。だからこそこれまでに、いろんな加速法が考案され、実用化されてきたのだ。（詳しくは、多田講師による講義を参照）

本講義では、時間の制約と講師のエネルギー不足の問題から、最も簡単な加速法の一つである　Successive Over Relaxation法（以下、SOR法）を取りあげて、外側反復における中性子束の外挿への適用を試みる。

3.4.1. SOR法の実装方針

ｎ回目の外部反復の中性子束 $\phi_{g,i}^{n}$ を n-1 回目の外部反復の中性子束 $\phi_{g,i}^{n-1}$ を用いて次式により外挿する。

$$\phi_{g,i}^{\prime n} = \phi_{g,i}^{n} + \omega(\phi_{g,i}^{n} - \phi_{g,i}^{n-1})$$

この実現に必要な基本的な処理は

- 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

の二つだけである。前回の中性子束を保存する場所は、Node クラスが妥当であろう。個々のノードに対して save_flux_old メソッドを追加し、外側反復のエネルギーループの最後の処理として Container クラスを通じて呼び出せば良いので、それほど難しくはなさそうだ。

3.4.2. 実装例

今回は、最もトップレベルの CalculationManager のテストコードから考えてみる。加速係数 ω を指定する必要があるので、CalculationManager のコンストラクタに与える config にその情報を含めた。asymptotic_criteria は収束が漸近状態に入っているかを判定するためのパラメータとして追加している。

test_calculation_manager.py

```
import unittest

import sys
sys.path.append('../lib')
import math

from config import *
from cross_section import CrossSection
from node import Node
from container import Container
from container_controller import ContainerController
from calculation_manager import CalculationManager

class CalculationManagerTest(unittest.TestCase):

    def test_calculation_manager(self):

        xs_fuel = CrossSection()
        xs_fuel.set([[1.58, 0.0032, 0.0, 1.0],[0.271, 0.0930, 0.168, 0.0]])
        xs_fuel.set_smat( [[0.0, 0.0178], [0.0, 0.0]])
        xs_fuel.calc_sigr()

        xs_ref = CrossSection()
        xs_ref.set([[1.41, 0.0, 0.0, 1.0],[0.117, 0.0191, 0.0, 0.0]])
        xs_ref.set_smat( [[0.0, 0.0476], [0.0, 0.0]])
        xs_ref.calc_sigr()

        geom = [{'xs':xs_ref, 'width':30}, {'xs':xs_fuel, 'width':60}, {'xs':xs_ref,
'width':30} ]

        delta = 1.0
        albedo = -1.0

        config = { 'geometry':geom, 'mesh_width':delta, 'albedo': albedo,
'max_iteration': 1000, \
                    'omega': 0.5, 'asymptotic_criteria': 0.05}

        calc_man = CalculationManager(config)
        count, flag = calc_man.run()
        print( "outer iterations:", count)

        keff = calc_man.get_keff()


        self.assertAlmostEqual(keff, 1.35826, places=5)  # keff with strict condition


if __name__ == '__main__':
    unittest.main()
```

このテストコードを成功させるために、CalculationManager, ContainerController, Container, Node をそれぞれ拡張していく。

以下に、各クラスの全体もしくは拡張部分の例を示す。

calculation_manager.py

```
from container import Container
from container_controller import ContainerController


class CalculationManager:
```

```
    def __init__(self, param):
        geom = param['geometry']
        delta = param['mesh_width']
        albedo = param['albedo']
        omega = param['omega']
        asym_cri = param['asymptotic_criteria']

        container = Container(geom, delta, albedo)
        container.set_omega(omega, asym_cri)

        self.controller = ContainerController(container)

    def run(self):
        return self.controller.calc()

    def get_keff(self):
        return self.controller.get_keff()
```

container_controller.py

```
class ContainerController:
    """
    Container
    """

    def __init__(self, container=None):
        self.cont = container
        self.keff = 1.0
        self.keff_old = 1.0
        self.total_fis_src_old = 1.0
        self.conv_criterion = 1.0E-7
        self.max_outer_iterations = 500
        self.inner_iterations = 4
        self.converged = False

    def calc(self):
        for idx_outer in range(self.max_outer_iterations):

            self.total_fis_src = self.cont.get_total_fis_src()
            norm_factor = 1.0 / (self.total_fis_src/self.keff)
            self.cont.normalize_fis_src(norm_factor)

            for kg in range(self.cont.get_ng()):
                self.cont.calc_scat_src(kg)

                for idx_inner in range(self.inner_iterations):
                    for color in range(2):
                        self.cont.calc(kg, color)

                    self.cont.calc_fis_src(kg)

                if self.cont.is_asymptotic(kg):
                    self.cont.accel_flux(kg)
                self.cont.save_old_flux(kg)

            self.total_fis_src = self.cont.get_total_fis_src()

            self.keff = self.total_fis_src / (self.total_fis_src_old/self.keff_old)
            diff = abs((self.keff - self.keff_old)/self.keff)

            #print(self.keff, diff)

            if(diff < self.conv_criterion):
                self.converged = True
                break

            self.keff_old = self.keff
            self.total_fis_src_old = self.total_fis_src

            self.cont.set_keff(self.keff)

        return (idx_outer, self.converged)

    def get_keff(self):
        return self.keff
```

container.py

```
import numpy as np

from node import *
from cross_section import *


class Container:
    def __init__(self, geometry, delta=1.0, albedo=0.0):
        self._setup(geometry, delta, albedo)

    def _setup(self, geometry, delta, albedo):
        self.nodes = []
        for r in geometry:
            for k in range(int(r['width']/delta)):
                the_node = Node(r['xs'])
                the_node.set_width(delta)
                self.nodes.append(the_node)

        self.delta = delta
        self.albedo = albedo
        self.ng = self.nodes[0].get_xs().ng()
        self.change_ratio_old = 1.0
        self.max_diff_old = 1.0
        self.asymptotic_criteria = 0.0

    (    )

    def accel_flux(self, kg):
        for the_node in self.nodes:
            the_node.accel_flux(kg)

    def save_old_flux(self, kg):
        for the_node in self.nodes:
            the_node.save_old_flux(kg)

    def set_omega(self, omega, asym_criteria=0.05):
        self.asymptotic_criteria = asym_criteria
        for the_node in self.nodes:
            the_node.set_omega(omega)

    def is_asymptotic(self, kg):
        diff = np.zeros(len(self.nodes))
        for i in range(len(self.nodes)):
            diff[i] = self.nodes[i].get_flux_diff(kg)
        max_diff = max(diff)

        change_ratio = max_diff / self.max_diff_old
        if change_ratio / self.change_ratio_old < self.asymptotic_criteria:
            flag = True
        else:
            flag = False

        self.change_ratio_old = change_ratio
        self.max_diff_old = max_diff

        return flag
```

node.py

```
import numpy as np

from cross_section import *
from config import *

class Node:
    def __init__(self, xs=None, kg=2):
        self.jout = np.ones((kg,2))   # kg, out-going, [XM, XP]
        self.jin  = np.ones((kg,2))   # kg, in-coming, [XM, XP]
        self.flux  = np.ones(kg)      # average flux
        self.old_flux = np.ones(kg)
        self.omega = 0.5
```

```
        self.width = 1.0
        self.xs = None
        self.keff = 1.0
        self.fis_src = np.ones(kg)
        self.scat_src = np.zeros(kg)
        if(xs):
            self.set_xs(xs)



    # SOR acceleration
    def save_old_flux(self, kg):
        self.old_flux[kg] = self.flux[kg]

    def accel_flux(self, kg):
        self.flux[kg] = self.flux[kg] + self.omega * (self.flux[kg] -
self.old_flux[kg])

    def set_omega(self, omega):
        self.omega = omega

    def get_flux_diff(self, kg):
        return (self.flux[kg] - self.old_flux[kg])/self.old_flux[kg]
```

3.4.3. 演習問題

1. 最も効率が良い加速因子 ω と、その時の収束回数を求めよ
2. メッシュ幅を変化させたとき、収束状況がどのように変化するかを考察せよ。