コンピュータ上のグラフ理論の進化

言語的能力と数学的能力の統合の可能性を考える

Agenda
コンピュータ上のグラフ理論の進化

- Part 1　はじめに
  – 「AIとグラフ」連続セミナーの流れ
- Part 2　グラフとAIのSynchronicity
- Part 3　グラフのカテゴリー論的特徴づけ　C-set
- Part 4　物理シミレーションとグラフ

Agenda Part 1
はじめに –「AIとグラフ」連続セミナーの流れ

- 前回のセミナー「AIとグラフ」の振り返り
- 連続セミナーの基本的な問題意識
  言語的能力と数学的能力の統合の可能性を考える
- 今回のセミナーの構成
- 今後のセミナーの展開予定

Agenda Part 2
グラフとAIのSynchronicity

- はじめに –グラフとAIのSynchronicity
- グラフ・ライブラリーの変化
  - Pyplot
  - DOT言語とGraphviz
  - NetworkX
  - Catlab.jl

Agenda Part 3
グラフのカテゴリー論的特徴づけ C-set

- Catlab とは何か？
- C-Setとはなにか？
  - グラフをFunctorで捉える
  - CatlabでのC-setプログラミング
  - 無向グラフと反射的グラフのスキーマ

Agenda Part 4
物理シミレーションとグラフ

- AlgebraicJuliaとは何か？
- 「AIと物理」というテーマの重要性
- Decapodes プロジェクトとは何か
- 単純な調和振動子の式をグラフで表す
- “A diagrammatic view of differential equations in physics”

セミナー関連リンク

- マルレク＋MaruLaboページ
  「コンピュータ上のグラフ理論の進化 」
  https://www.marulabo.net/docs/ai-graph2/

- YouTube Maruyama Lectures
  「コンピュータ上のグラフ理論の進化 」再生リスト
  https://www.youtube.com/playlist?list=PLQIrJ0f9gMcMIF7gvAb6Xp1kiMM6X5woq

Part 1

はじめに –「AIとグラフ」連続セミナーの流れ

Agenda Part 1
はじめに –「AIとグラフ」連続セミナーの流れ

- 前回のセミナー「AIとグラフ」の振り返り
- 連続セミナーの基本的な問題意識
  言語的能力と数学的能力の統合の可能性を考える
- 今回のセミナーの構成
- 今後のセミナーの展開予定

前回のセミナー「AIとグラフ」の振り返り

前回のマルレク「AIとグラフ -- GPT4oで遊ぶ」では、画像の認識と生成を新しい特徴とする「マルチ・モーダルAI」のGPT4oが、画像としてのグラフの生成ではつまずくということから、画像とグラフの違いやマルチ・モーダルなAIの発展の方向を考えてみたものです。


それは、ことばや絵より難しいんだ。
大人だって、そうなんだよ。
ぼく、グラフうまく描けないみたい。

ただ、前回のセミナーは、問題提起はしたものの 議論の展開の途中で終わっています。今回以降のセミナーで、いくつかの残された課題に応えていこうと思っています。

グラフの複雑さと対話型証明

前回のセミナーのPart 2「グラフの認識の難しさ」では、複雑性理論からグラフの認識の難しさを論じました。

そこで紹介した「対話型証明」という新しい証明手法は、証明者中心で決定論的な従来の証明観から、検証者中心で確率論的な証明観への変更をもたらすものです。

こうした変化を理論的に捉えるには、量子論の助けが必要なのですが、セミナーではその点には触れられませんでした。

ただ、「対話型証明」の理解については、量子コンピュータとその「量子複雑性」の理論を持ち出さなくても、面白いことが起きていると思います。

セミナーの中で触れたのですが、「対話型証明」のかつては一番奇妙に思えた想定、「万能だが嘘をつくことがある全能者」の存在についての感覚的な理解は、むしろ広がっています。なぜなら、「AIは万能だ」と考える人が増えている一方で、「AIが真っ赤な嘘をつく」ことを、多くの人は体験しているからです。

理性的で批判的な「検証者」の役割を、我々人間が誠実に果たすことが求められていることが、「対話型証明」の真骨頂だと僕は考えています。

証明者。全知全能である。
どんな問題も瞬時に答えを返す能力をもつ。ただし、時々嘘をつく。

IP

Interactive Proof

二人が、対話を繰り返すと
何が証明できるか？

Question

Answer

検証者。普通の人間である。
理性をもっていて、証明者の主張を検証しようとする。盲信はしない。

生成AI。
どんな問題も瞬時に答えを返す能力があるように見える。
ただし、時々嘘をつくことは、多くの人は気がついている。

Answer

Question

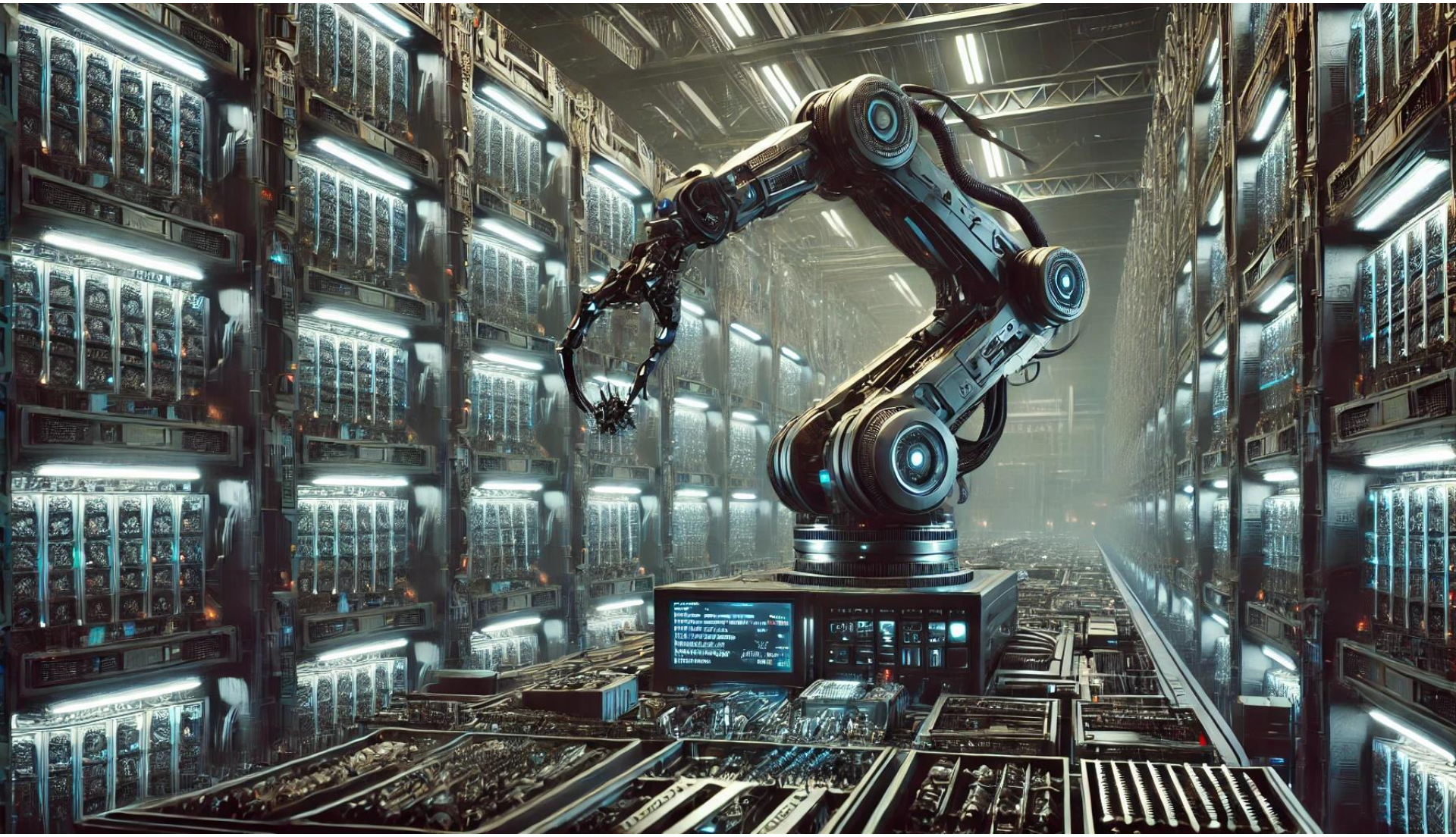
生成AIのもとでの我々の経験

我々。普通の人間である。
理性をもっていて、生成AIの主張を検証しようとする。盲信はしない。

Agent-Based Model について

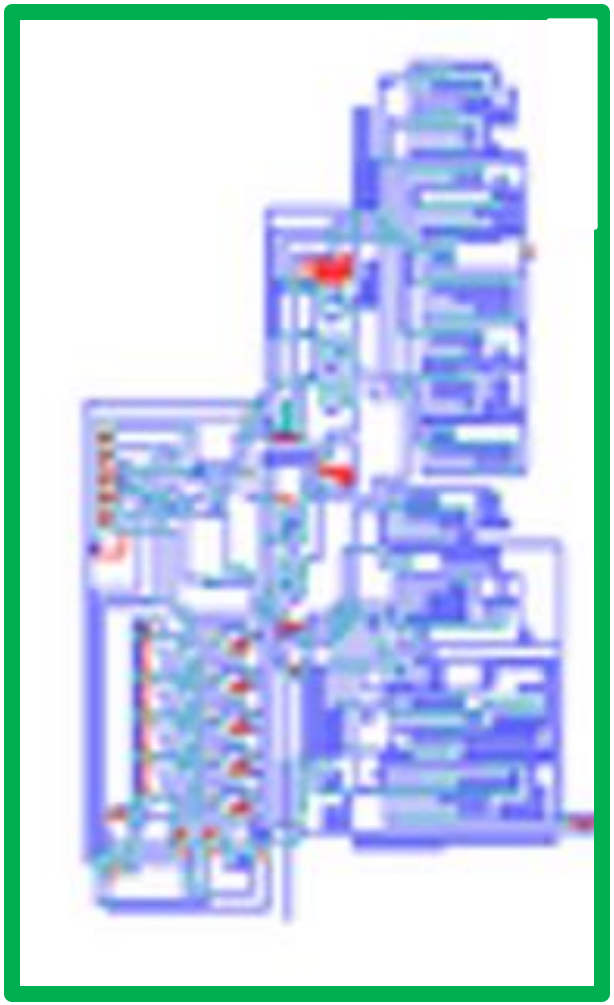
前回のセミナーのPart 3「Agent-Based Model について」では、独立した主体としてのAgentが、「外部」の「環境」から情報を入力として受け取り、Agentが内部で処理した結果を、「外部」に「アクション」あるいは情報を出力として返すAgentモデルを紹介しました。

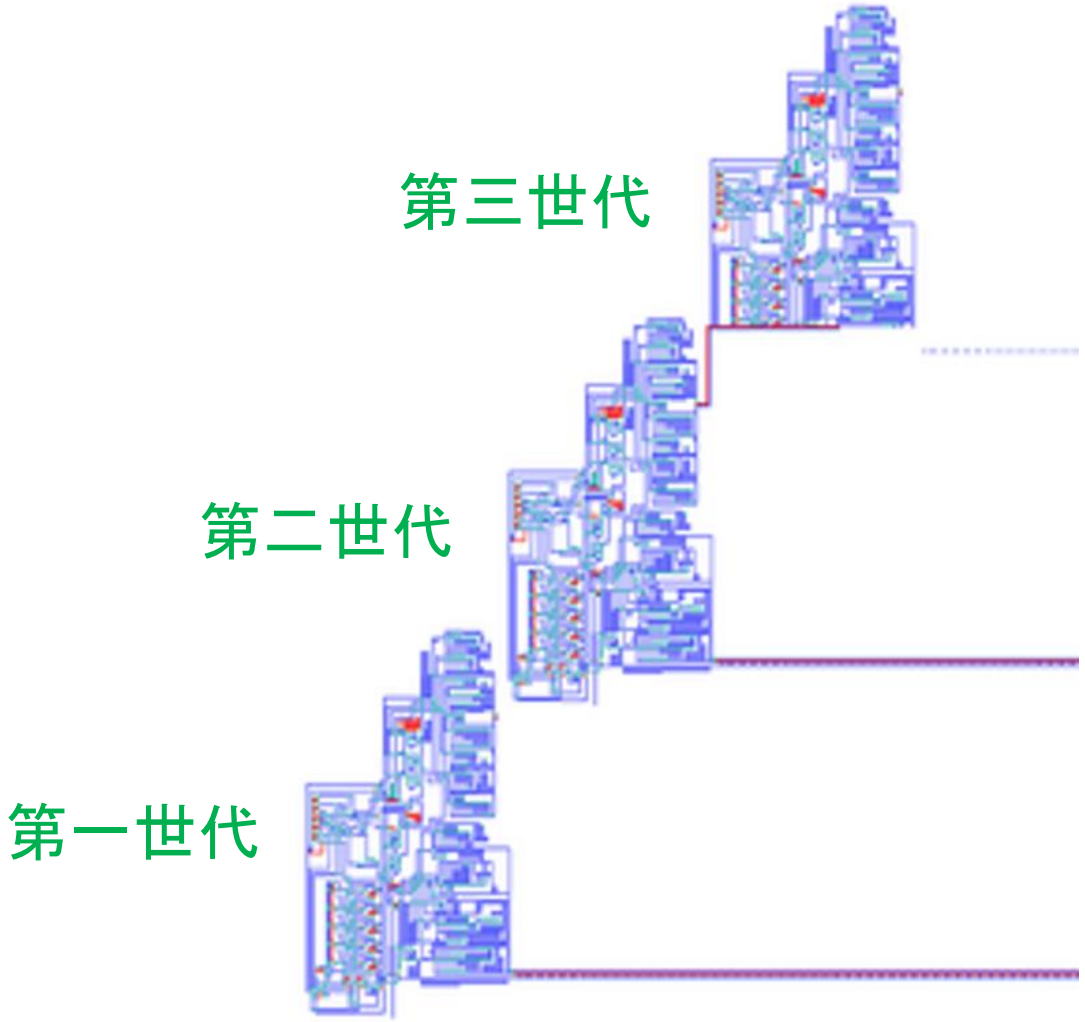
Von Neumann's kinematic model



フォン・ノイマンのセルラー・オートマトン・モデル

複製されるべきパターン





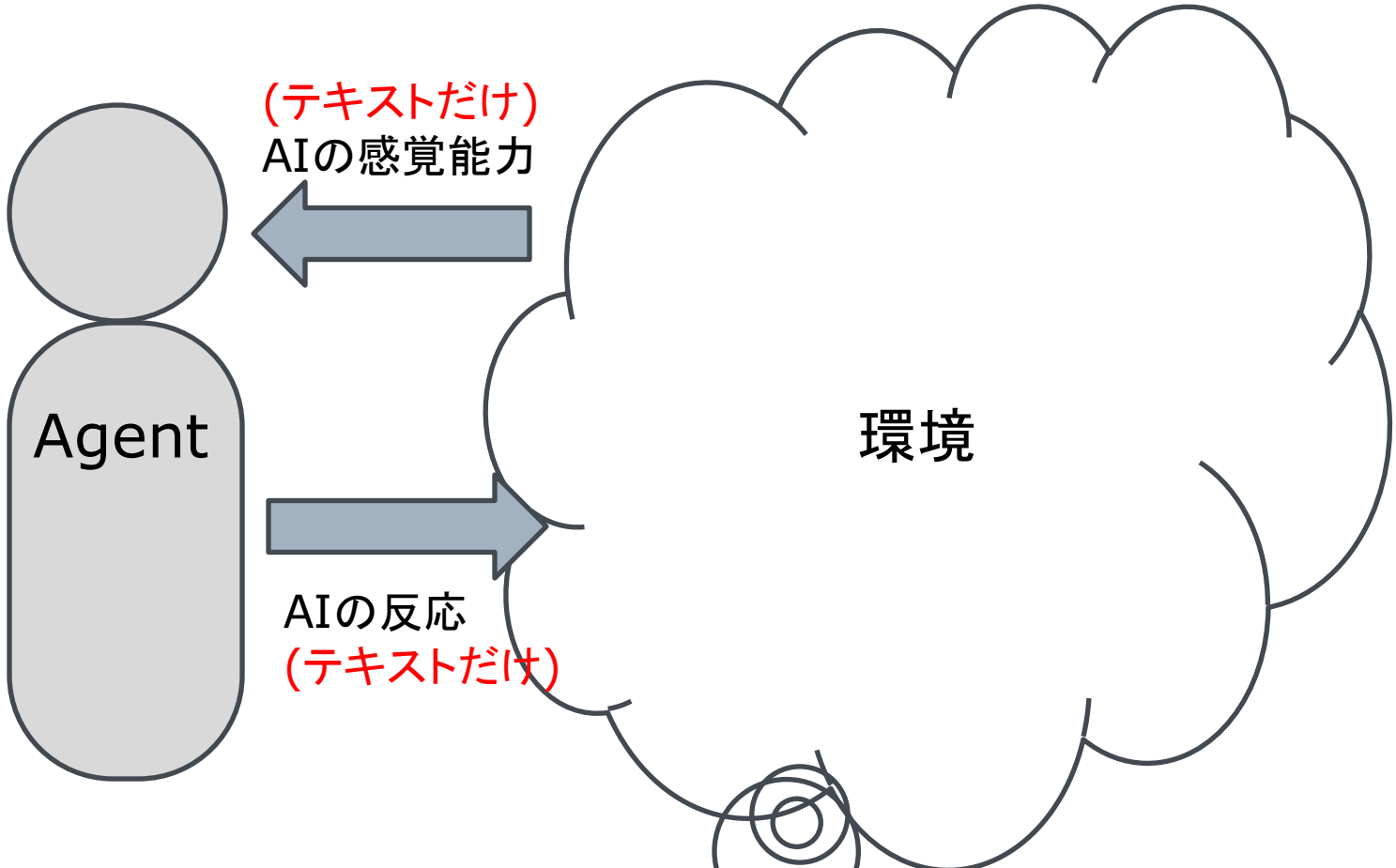
Agent-Based Model が注目される理由

AIの世界でAgentモデルが注目されているのは、基本的には次のような理由だと僕は考えています。

一つには、AIのマルチ・モーダル化の流れは、AIと「外部」との接点が、これまでの言語情報の交換という制約をもつLLMを超えて、今まで以上に多様化し拡大することを意味します。

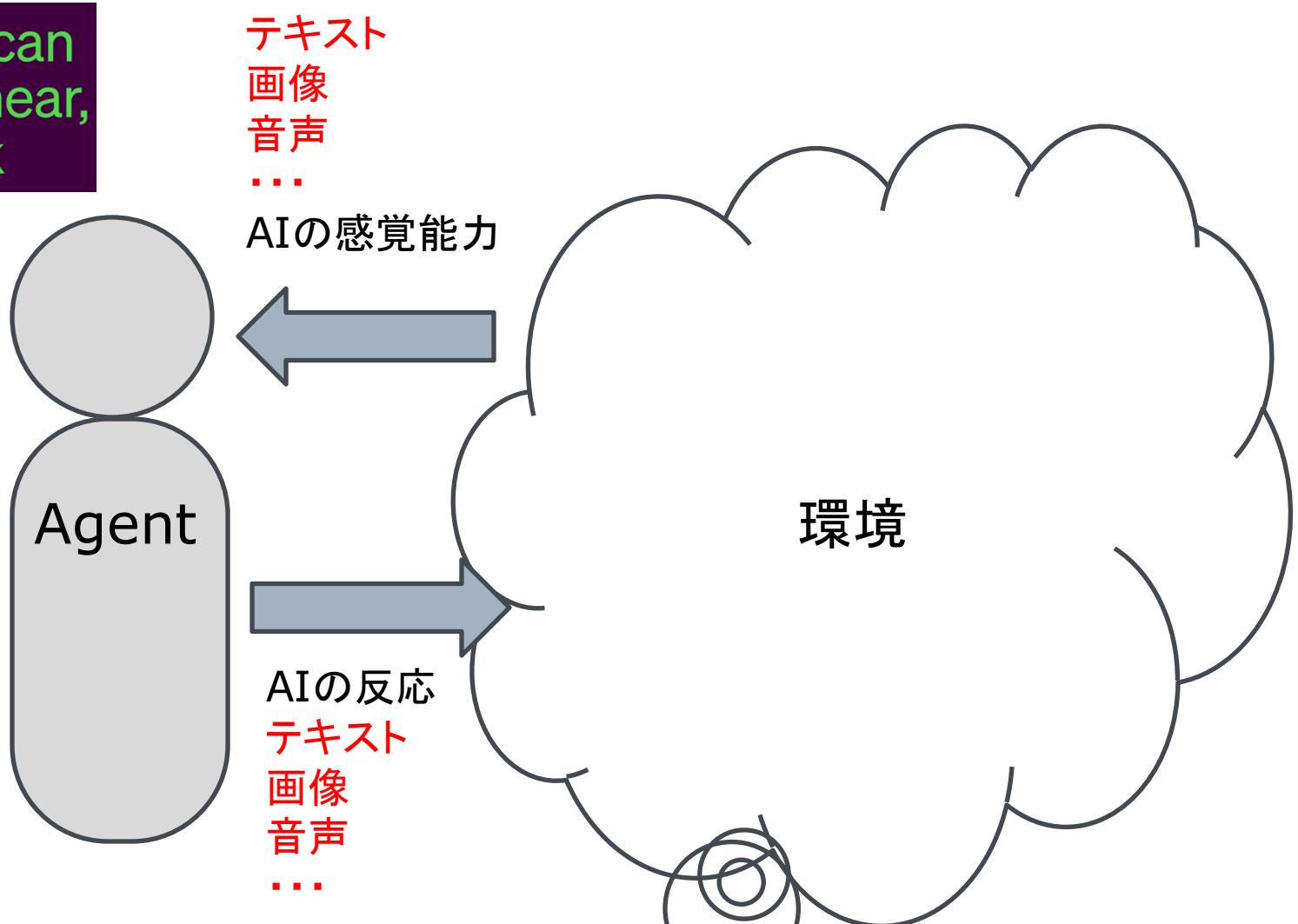
AIと自律型ロボットとの境界がだんだん低くなっていくと思います。そして、そうした展望は、AIを自律的Agentとして捉えるというAIの長期的発展の展望と一致しています。

マルチモーダル以前のAIのモデル



マルチモーダルなAIのモデル
Agent Base Model

ChatGPT can now see, hear, and speak



Agent-Based Model が注目される理由

もう一つは、我々人間は、抽象的には自律的Agentと考えることができます。客観的には、我々はAIの「外部」にあります。外部から現在のAIの到達点を評価して、その変化の可能性を予測し、シミレートするモデルが必要になっていると僕は考えています。

先月のセミナーで紹介した、RAGやATMやAgentGymといったLLMの拡大の試みは、いずれも現在のLLMを一つのAgentとして捉え返すものとして解釈可能です。

ただ、それらは現在のAIの枠内での試みです。複数のAgent(多数の人間・多数のAI)が「外部」と相互作用する Multi-Agent System 、人間とAIとその「外部」が相互作用するシステムを考えることができます。

こうしたAgentモデルの「観測」と「作用」の数学的モデルは、物理学のモデルも提供することになると僕は考えています。

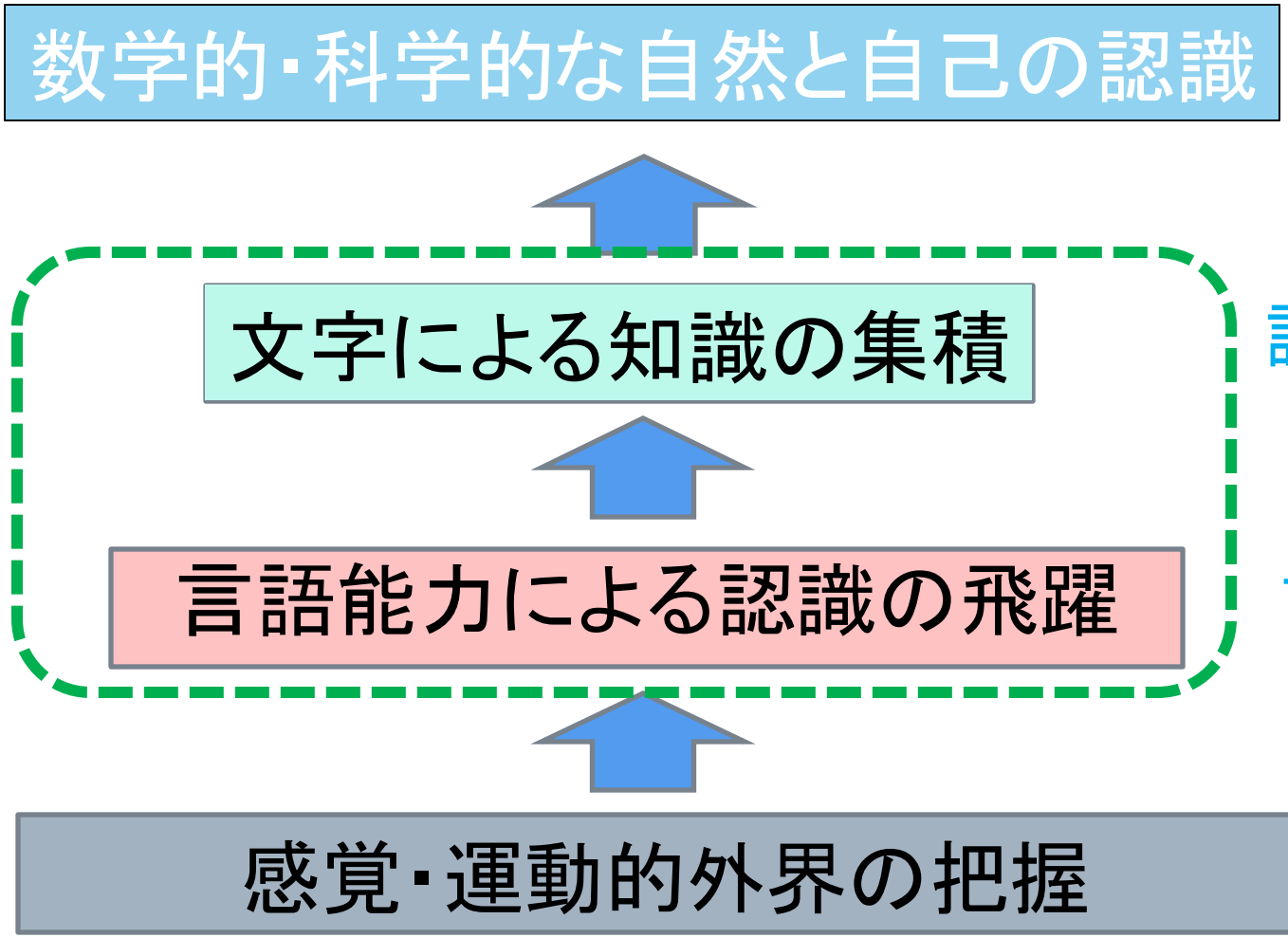
連続セミナーの基本的な問題意識
言語的能力と数学的能力の統合の可能性を考える

現在のLLMベースのAIに数学的能力を統合したいというのが僕の大きな問題意識なのですが、話が、少し広がりすぎました。それでも、まだまだ考えなければいけないことがたくさんあります。二つの能力の統合には、一定の時間が必要だと思います。

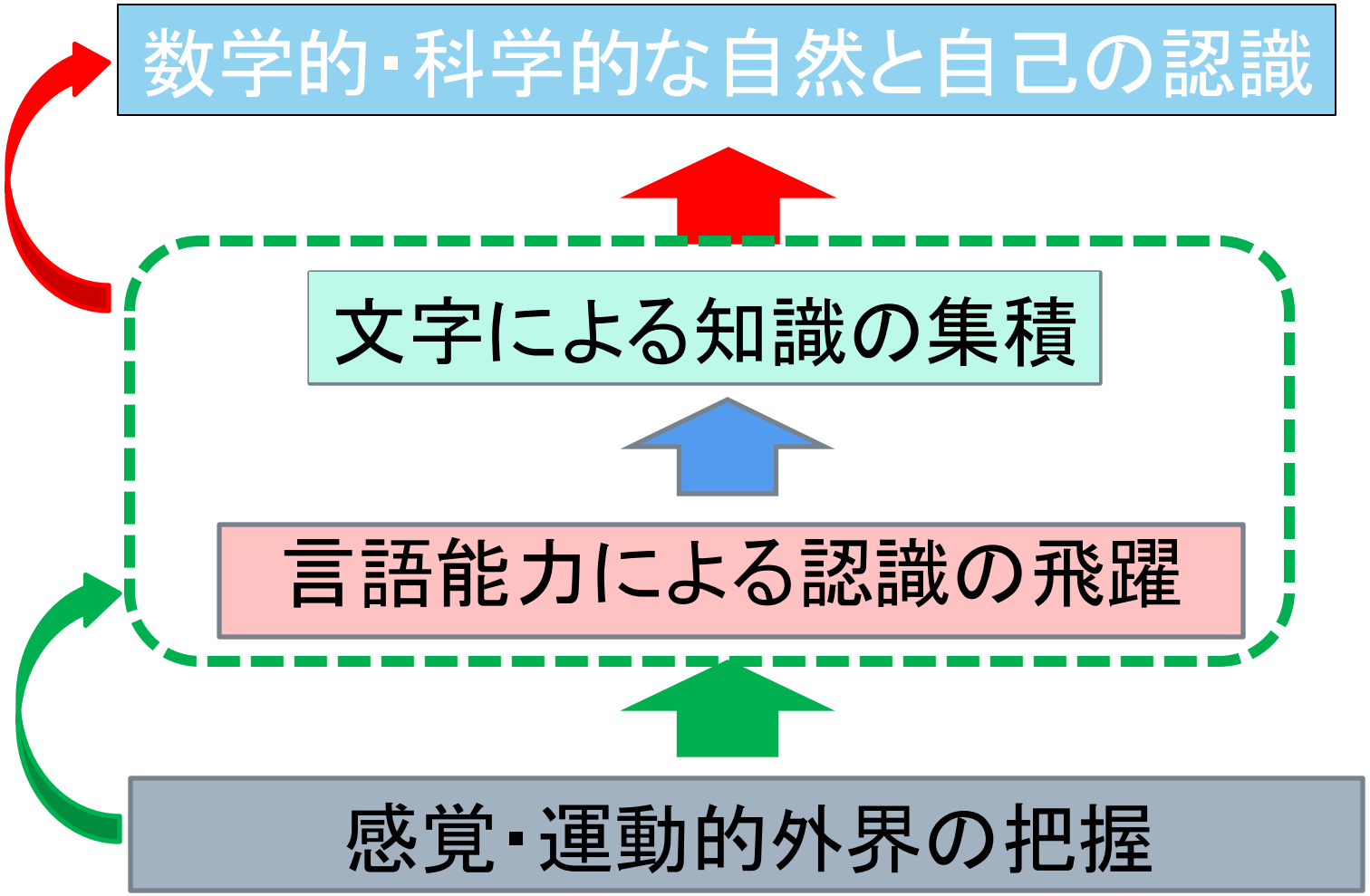
ただ、言語のモデルと数学のモデルとが統合されうるのであれば、両者はある共通の特徴を持っているはずです。そうした特徴は、カテゴリー論によって与えられると考えています。

当面、グラフとAgent モデルの二つを糸口にして、そうした共通の特徴を考えることから始めていきたいと思います。

人間の認識の発展と階層



人間の認識能力の
大きな飛躍は二つある



反省していること

もともと今回のセミナーは、「AIの言語的能力と数学的能力の統合の可能性を探る」と、大きな風呂敷を広げていたのですが、そうした議論をほとんど展開できないままになってしまいました。

特に、C-setとしてのグラフを生み出すfunctorをオブジェクトとし、functor間のnatural transformationを射とするグラフのカテゴリーとしてのfunctor カテゴリー C-Setについては説明することができませんでした。

重要なことは、このC-Setがカテゴリー論的にはco-presheaf であることです。

近年のグラフについてのこうした認識の深化が、昨年末から紹介してきた Tai-Danae Bradleyらの大規模言語モデルの「co-presheaf 意味論」と「共鳴」しているのだというビジョンを僕は持っています。

ただ、理論的な展開や歴史的な認識の媒介が欠けたままでは、それは、ほとんど僕のハルシネーションのままです。（presheafもco-presheafも、数学的には至る所に存在します。
「Syncronicity」は、「統合」とは違うものです。ま、それはしょうがないにしても。）

語れなかったことが多かったのですが、今回のセミナーは、今後の展開への基礎になると思います。特に、Part 2 とPart 3は、Tutorial的な内容で、わかりやすいものだと思います。

今回のセミナーの構成

今回のセミナー「コンピュータ上のグラフ理論の進化」は、次のような構成を考えています。

- Part 1　はじめに – 「AIとグラフ」連続セミナーの流れ
- Part 2　グラフとAIのSynchronicity
- Part 3　グラフのカテゴリー論的特徴づけ C-set
- Part 4　物理シミレーションとグラフ

Part 2　グラフとAIのSynchronicity

Part 2　「グラフとAIのSynchronicity」では、NetworkX, Graphviz, Catlabという代表的な三つのグラフ・ライブラリーを取り上げ、その特徴とその 実装の変化を見ていきます。

基本的な問題意識は、6月のマルレク「AIとグラフ」で見たように、現在のAI技術が「数学的なもの」の認識に弱点をもつことの背景を、グラフの本質把握の未熟さと通底する問題として考えようと言うことです。

AI技術の変化に暗に対応するように（実は、相互に独立なのですが）、従来のグラフ・ライブラリーの弱点を補う新しいグラフ・ライブラリーが登場しています。

Part 3 「グラフのカテゴリー論的特徴づけ C-set」

Part 3 「グラフのカテゴリー論的特徴づけ C-set」は、今回のセミナーの中心的な内容になります。そこでは、グラフにC-setというカテゴリー論的な特徴づけが与えられます。

残念ながら、今回のセミナーのコンテンツは、そこで終わっています。

Part 4　物理シミレーションとグラフ

僕は、大規模言語モデルベースのAI技術では、数学だけではなく「物理シミレーション」の世界にも届かないだろうと考えています。それは、現代のAI技術と現代の科学の間には、埋められるべき広大な空白があることを意味します。

Part 4「物理シミレーションとグラフ」では、科学・技術の世界でのコンピュータ利用の重要な分野である「物理シミレーション」へのグラフの応用として、AlgebraicJuliaのDecapodesプロジェクトを紹介しています。

驚くべきことに、そこでは基本的な微分方程式間の関係を定義するグラフから、物理シミレーションのコードを作成するというアプローチが取られています。

今後のセミナーの展開予定

語れることは語った方がいいと思います。基本的には、今回十分な展開ができなかった「AIの言語的能力と数学的能力の統合の可能性を探る」というテーマを継続して考えていきたいと思っています。

その上で、いくつか展開すべき問題が残されています。

ひとつは、「コンピュータと数学」というテーマです。

そもそも、コンピュータには、どのような「数学的能力」があるのかという話です。それはまた、人間にはどのような「数学的能力」があるのかという話でもあります。歴史的な振り返りが中心になります。

もう一つは、理論的には、「Agent-Based Model」の数学的理論です。

現代のAgent-Based Modelは、ある意味で、機械の「数学的能力」を代表するTuringマシンやvon Neumann の自己複製マシンの拡張として解釈できます。しかも、決定論的ではなく確率論的拡張として。そのうえ、それは「外部の世界」を意識しています。

こうした方向については、「今月のセミナーについて」という形で、公開したいと思います。

セミナー「コンピュータと数学」について

次のような内容で、セミナー「コンピュータと数学」を開催したいと思っています。

セミナー「コンピュータと数学」

- Turingマシンの数学的射程
- 量子コンピュータと量子複雑性理論
- Baezの「Rosetta Stone論文」
- Voevdskyの数学でのコンピュータ利用のビジョン
- Baezのネットワーク論とco-span
- CoqとGATlab

セミナー「Agent-Based Modelの数学的理論」

先のような問題の解説が一通り終わったら、「Agent-Based Modelの数学的理論」というセミナーを開催したいと思っています。


Part 2
グラフとAIのSynchronicity

Agenda Part 2
グラフとAIのSynchronicity

- はじめに –グラフとAIのSynchronicity
- グラフ・ライブラリーの変化
  - Pyplot
  - DOT言語とGraphviz
  - NetworkX
  - Catlab.jl

グラフとAIのSynchronicity

このセクション「**グラフとAIのSynchronicity**」は、現在のコンピュータのソフトウェアがグラフをどのように扱っているかを振り返ったものです。

次のようなグラフ・ライブラリーを取り上げています。

1. Pyplot
2. DOT言語とGraphviz
3. NetworkX
4. Catlab.jl

Synchronicity

これらの四つのセッションは、グラフ描画のライブラリーが、プリミティブなものからより洗練されたものに、どのように変化してきたのかがわかりやすいように配列されています。

重要なことは、こうした変化は、技術的な洗練にとどまらず、「グラフとは何か？」という基本的認識の理論的深化として捉えることができるということです。特に、最後に取り上げるAlgebraicJuliaのCatlabの技術的・理論的到達点は注目すべきものです。

非常に遠回りなアプローチですが、コンピュータ・サイエンスのある分野・グラフ描画の世界で起きている変化が、AI技術の未来の変化とシンクロしているのかもしれないと考えるのは、楽しいことです。

おわび

最初にお詫びしなければなりません。

セミナーに向けたセッションの中でで、DOT言語やGraphvizがNetworkXの後に生まれたように書いていたのですが、誤りでした。

MaruLaboのスタッフの一人から指摘がありました。彼は20年位前に、アプリの仕様設計で状態遷移図を描くのにGraphvizを使っていたそうです。

“The History of Graphviz” https://forum.graphviz.org/t/the-history-of-graphviz/765/1 、”Graphviz wiki” https://en.wikipedia.org/wiki/Graphviz によると、Graphvizの最初のリリースは、1991年以前だということです。

一方、NetworkXのリリースは、wikiによると 2002年に開発が始まり、最初のリリースが2005年です。

https://en.wikipedia.org/wiki/NetworkX

DOT 言語が、Graphviz プロジェクトの一部として開発されたのは確かなようで、2007年以前にMicrosiftのWordの記述言語に利用されていた.dot ファイルとの混同を避けるために、.gvという拡張子を使うようになったと、DOT Language wiki https://en.wikipedia.org/wiki/DOT_(graph_description_language) には記されています。

グラフ記述言語としては、2001年に開発が始まり2002年に最初のリリースが行われたXMLをグラフ記述に利用するGraphMLも利用されているようです。http://graphml.graphdrawing.org/

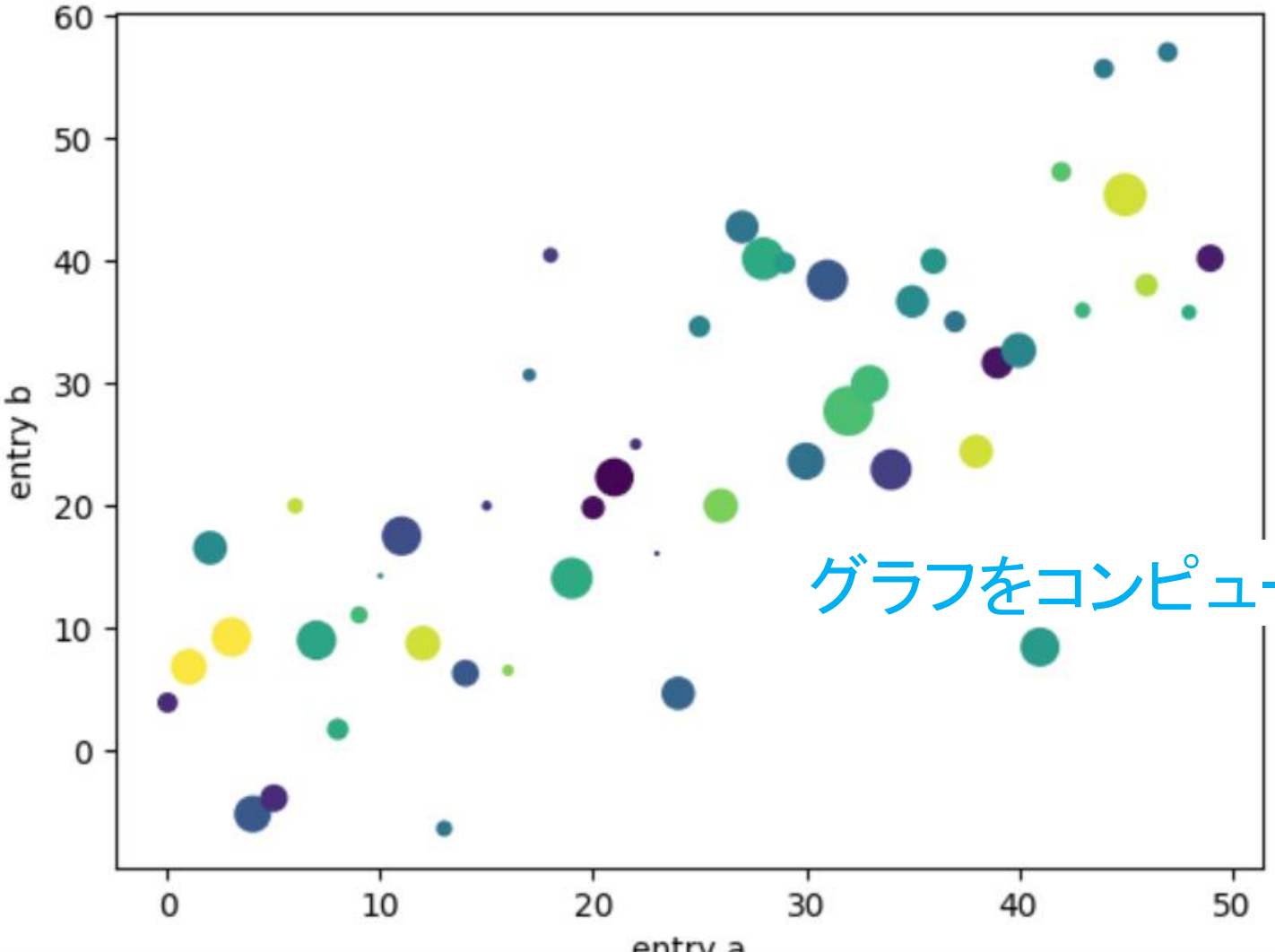
最初のリリースで並べると、次のようになります。

1991年　Graphviz + DOT Language
2002年　GraphML
2003年　Matplotlib
2005年　NetworkX
2019年　Catlab

最新のStableリリースは、次のようになります。

Graphviz　2024/04/28 Version 11.0.0
Matplotlib 2024/05/15 Version　3.9.0
NetworxX 2024/04/06 Version　3.3
Catlab　　2024/07/10 Version　0.16.15

PyPlot

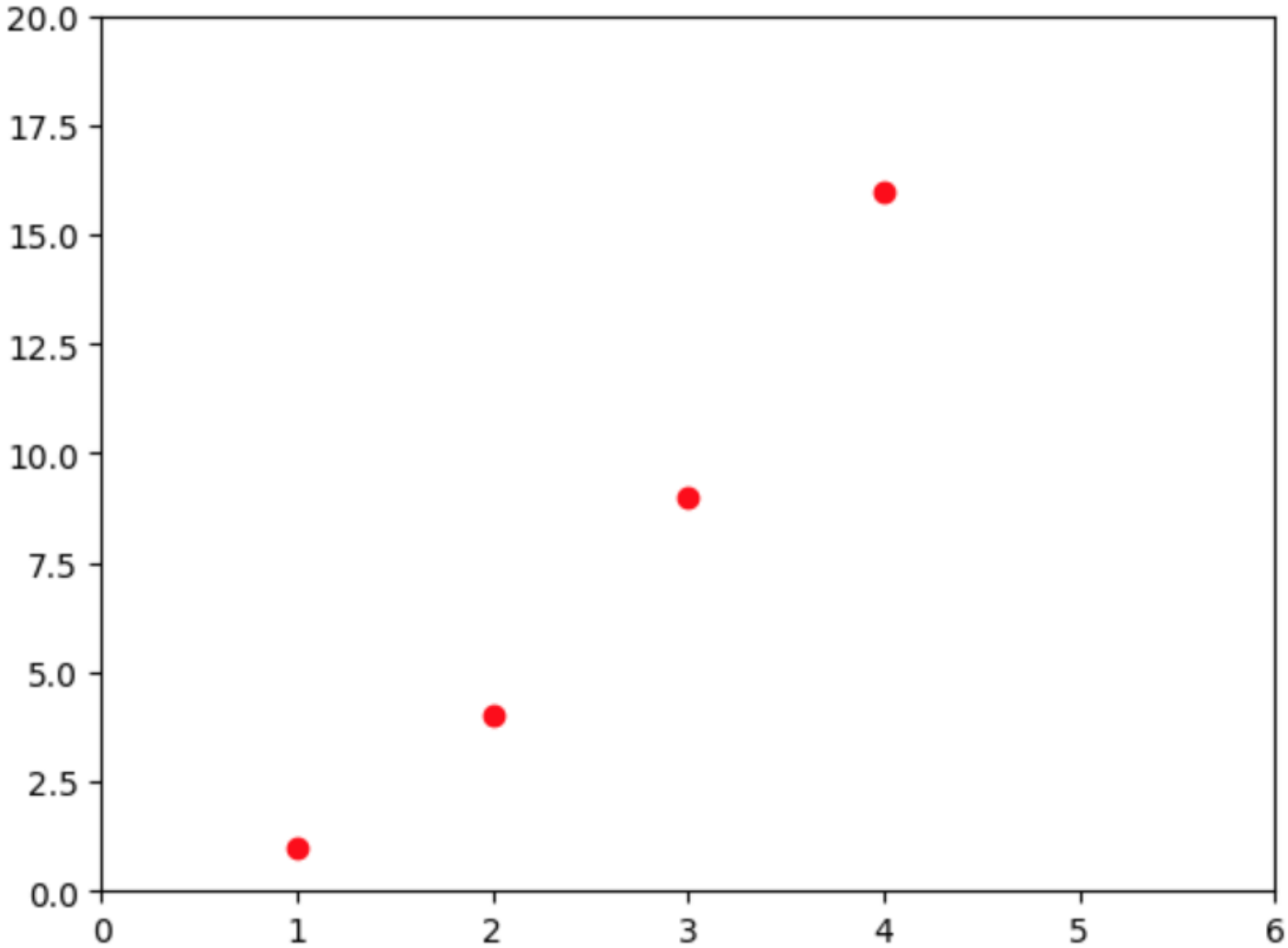


グラフをコンピュータで描く 1

Pyplot

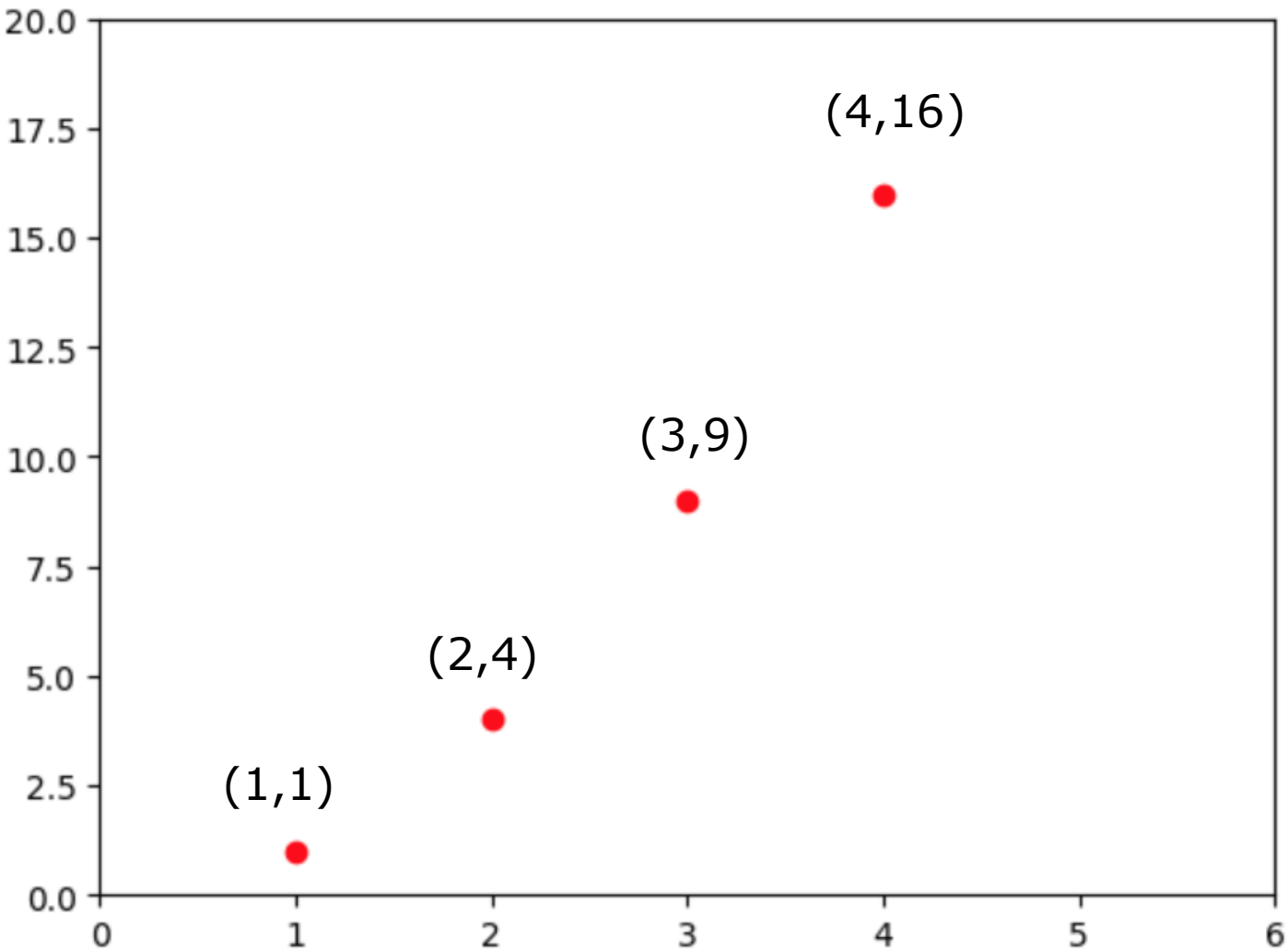
matplotlib.pyplot は、グラフ描画のライブラリーとして、
Pythonの初心者にもよく知られているもののーつです。

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'ro')
plt.axis((0, 6, 0, 20))
plt.show()
```



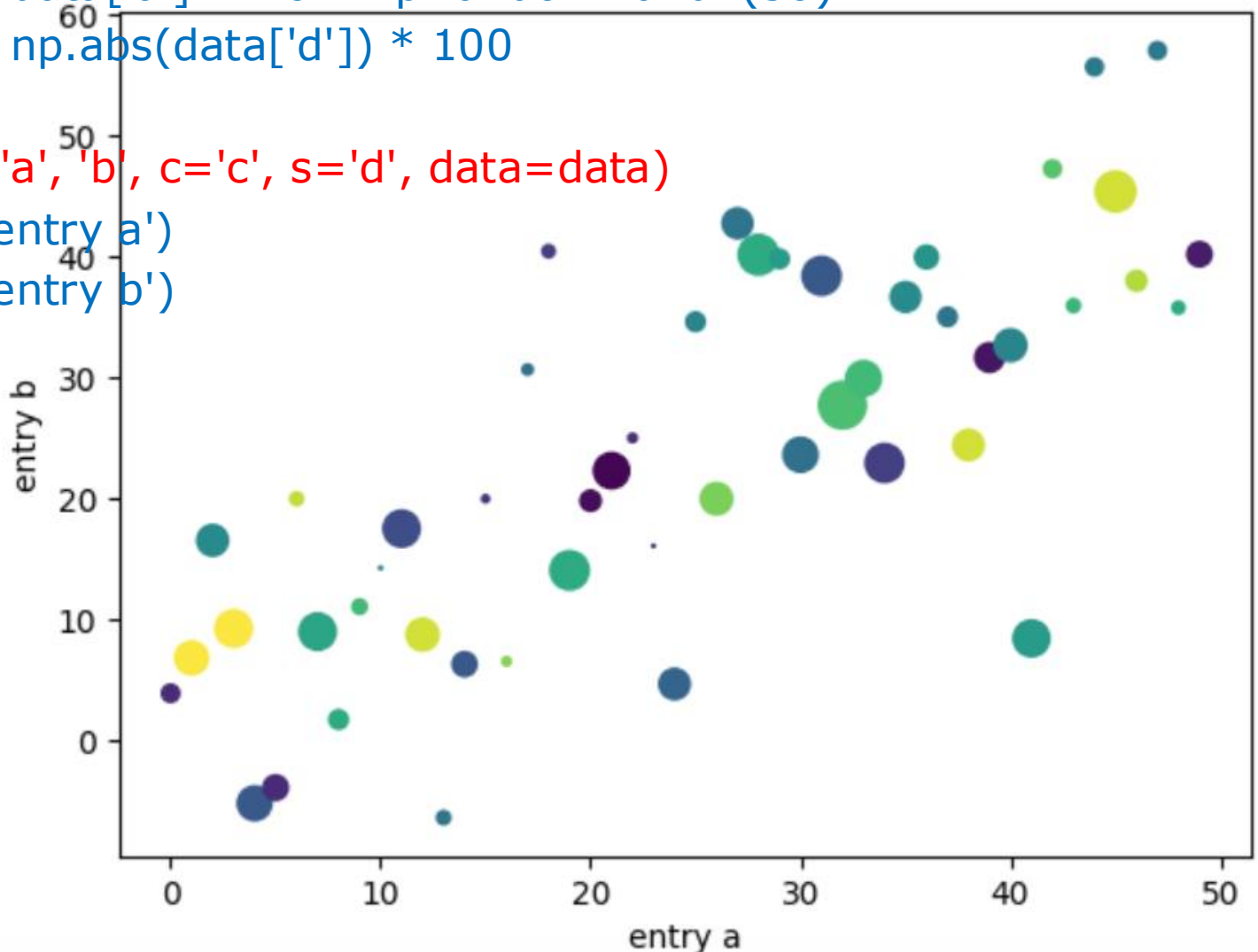
```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'ro')
plt.axis((0, 6, 0, 20))
plt.show()
```

‘ro’ は赤丸



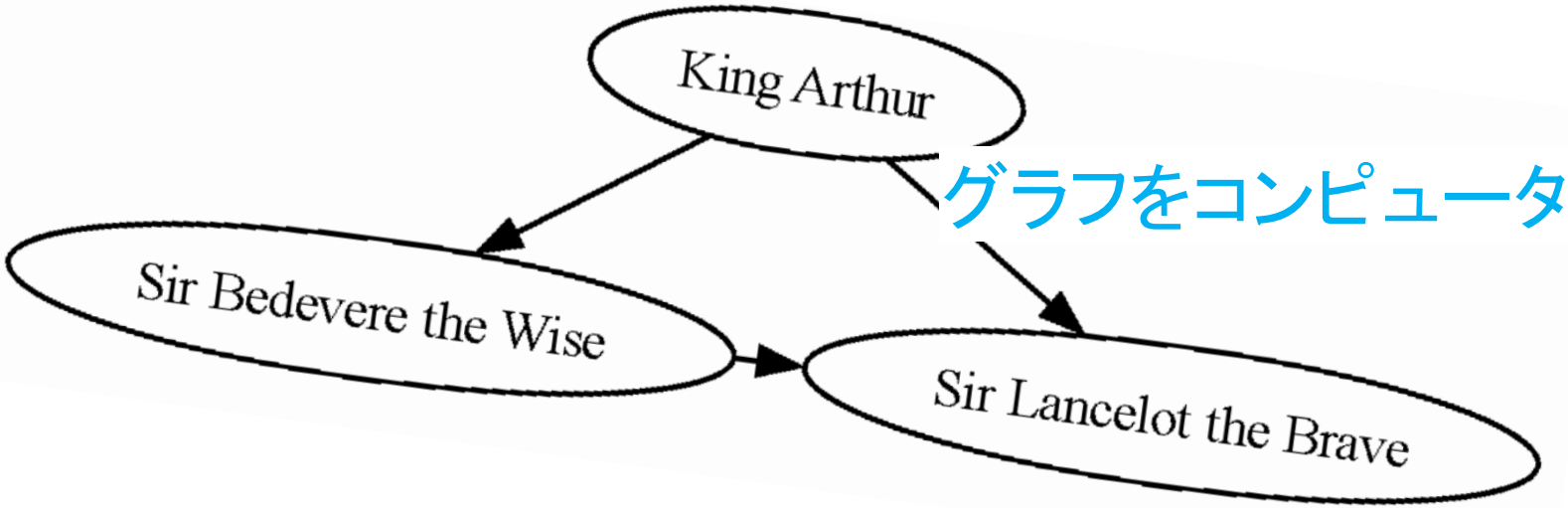
```
data = {'a': np.arange(50),
        'c': np.random.randint(0, 50, 50),
        'd': np.random.randn(50)}
data['b'] = data['a'] + 10 * np.random.randn(50)
data['d'] = np.abs(data['d']) * 100

plt.scatter('a', 'b', c='c', s='d', data=data)
plt.xlabel('entry a')
plt.ylabel('entry b')
plt.show()
```

```
data = {'a': np.arange(50),
        'c': np.random.randint(0, 50, 50),
        'd': np.random.randn(50)}
data['b'] = data['a'] + 10 * np.random.randn(50)
data['d'] = np.abs(data['d']) * 100

plt.scatter('a', 'b', c='c', s='d', data=data)
plt.xlabel('entry a')
plt.ylabel('entry b')
plt.show()
```



DOT言語とGraphviz

```
digraph "round-table" {
        A [label="King Arthur"]
        B [label="Sir Bedevere the Wise"]
        L [label="Sir Lancelot the Brave"]
        A -> B
        A -> L
        B -> L [constraint=false]
}
```



グラフをコンピュータで描く 2

グラフの「抽象的定義」と「描画」との分離

Graphvizでもっとも注目すべきことは、そこでは明確に、グラフの抽象的な「定義」とグラフの具体的な画像への変換である「描画」との分離が行われていることです。

それは、「抽象的な定義」で与えられる「抽象的なグラフ」と、画像としての「具体的なグラフ」を区別して考えようということです。

Graphvizでは、「グラフの抽象的な定義」を「DOT言語」という言語で行い、それを入力として受け取った描画エンジン(renderer)が、「具体的なグラフ」を出力するという二段階のアプローチを取ります。

最初は、「僕らは、頭の中のグラフのイメージを画像にするのに、コンピュータを使っているわけで、それは当たり前のことじゃない？ グラフ・ライブラリーの実装のレベルで、そういう区別を導入する と何かいいことがあるの？」と思われるかもしれません。

確かに、1時間ごとの温度の変化をPyPlotで折れ線グラフで表示しようというときには、そうした区別は、あまり意味がないかもしれません。

また、自分の組織のネットワーク図に、あたらしいノードを一つ追加しようとするなら、すでにあるNetworkXのプログラムに手を加えるのは簡単です。

先にみたように、PyPlotとNetworkXによるグラフ出力より、Graphvizが出力するグラフの表現力は高いものになっています。

話はすこし飛ぶのですが、頭の中の三角形と、コンピュータの描く三角形は違うものです。頭の中の円と、コンピュータが画面に出力する円の画像も違うものです。この例での「頭の中の図形」は、抽象的には「数学的な図形」です。

DOT言語とGraphvizは、グラフの中の世界でですが、初めてこうした道に踏み出したものだと思います。

そして、この道が大きな意味を持つことは、その後で登場したCatlabが、これらから何を継承したのかを見れば、さらに明確になります。

DOT言語

DOT言語は、グラフを抽象的に定義する言語である。

DOT言語で定義されたグラフは、renderer（描画プログラム）
dot の働きによって、具体的なグラフの画像に変換される。

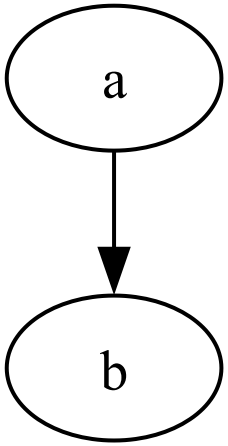
DOT言語によるグラフ定義と renderer dotによる画像出力

- 次のテキストは、DOT言語によるグラフ定義である。

```
digraph { a -> b }
```

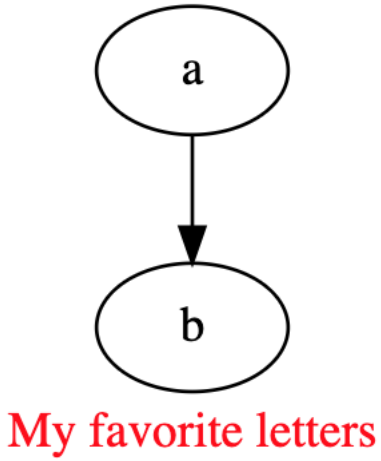
- 次の例は、renderer dot コマンドによる、標準入出力のパイプを通じた svg形式でのグラフ画像の出力である。

```
$ echo 'digraph { a -> b }' | dot -Tsvg > output.svg
```

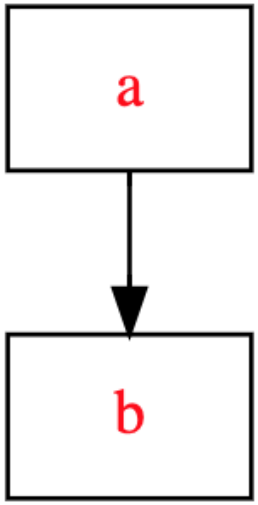


同じDOT言語でのグラフ定義でも、dot コマンドのパラメーターで画像を変えることができる

```
$ echo ‘digraph { a -> b }’ |
  dot -Tpng -Gfontcolor=red -Glabel="My favorite letters"
```



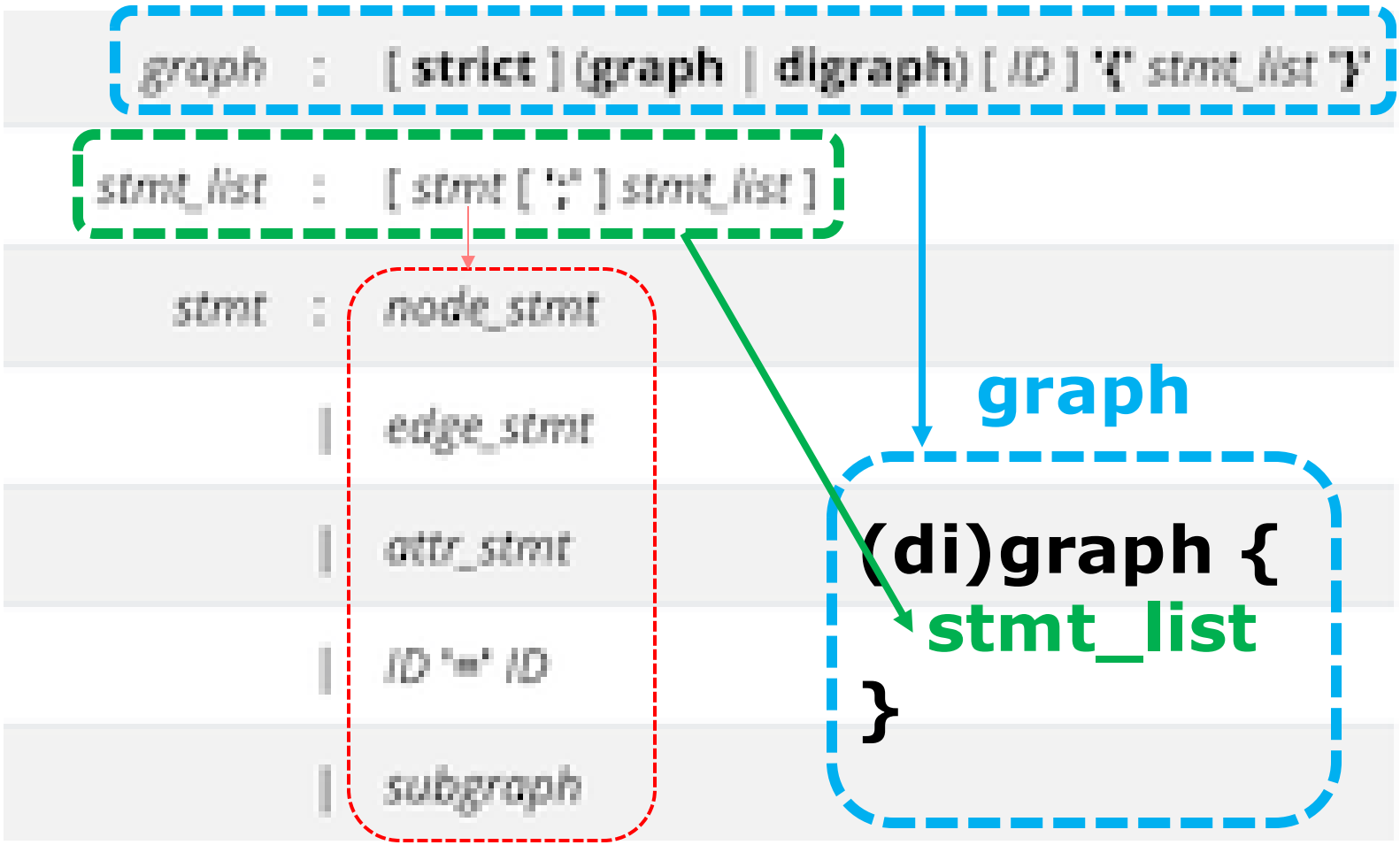
```
$ echo 'digraph { a -> b }' |
dot -Tjpg -Nfontcolor=red -Nshape=rect
```



DOT言語の仕様

https://graphviz.org/doc/info/lang.html

graph	:	[**strict**] (**graph** | **digraph**) [*ID*] '{' *stmt_list* '}'
stmt_list	:	[*stmt* [';'] *stmt_list*]
stmt	:	*node_stmt*
	|	*edge_stmt*
	|	*attr_stmt*
	|	*ID* '=' *ID*
	|	*subgraph*


graph : [strict] (graph | digraph) [ID] '{' stmt_list '}'
stmt_list : [stmt [';'] stmt_list]
stmt : node_stmt
| edge_stmt
| attr_stmt
| ID '=' ID
| subgraph
graph
(di)graph {
stmt_list
}

attr_stmt	:	(**graph** | **node** | **edge**) *attr_list*
attr_list	:	**'['** [*a_list*] **']'** [*attr_list*]
a_list	:	*ID* **'='** *ID* [(**';'** | **','**)] [*a_list*]
edge_stmt	:	(*node_id* | *subgraph*) *edgeRHS* [*attr_list*]
edgeRHS	:	*edgeop* (*node_id* | *subgraph*) [*edgeRHS*]
node_stmt	:	*node_id* [*attr_list*]
node_id	:	*ID* [*port*]
port	:	**':'** *ID* [**':'** *compass_pt*]

RHS: Right-hand side of an equation

```
attr_stmt   :   (graph | node | edge) attr_list
attr_list   :   '[' [ a_list ] ']' [ attr_list ]
a_list      :   ID '=' ID [ (';' | ',') ] [ a_list ]
edge_stmt   :   (node_id | subgraph) edgeRHS [ attr_list ]
edgeRHS     :   edgeop (node_id | subgraph) [ edgeRHS ]
node_stmt   :   node_id [ attr_list ]
node_id     :   ID [ port ]
port        :   ':' ID [ ':' compass_pt ]
```

属性リスト

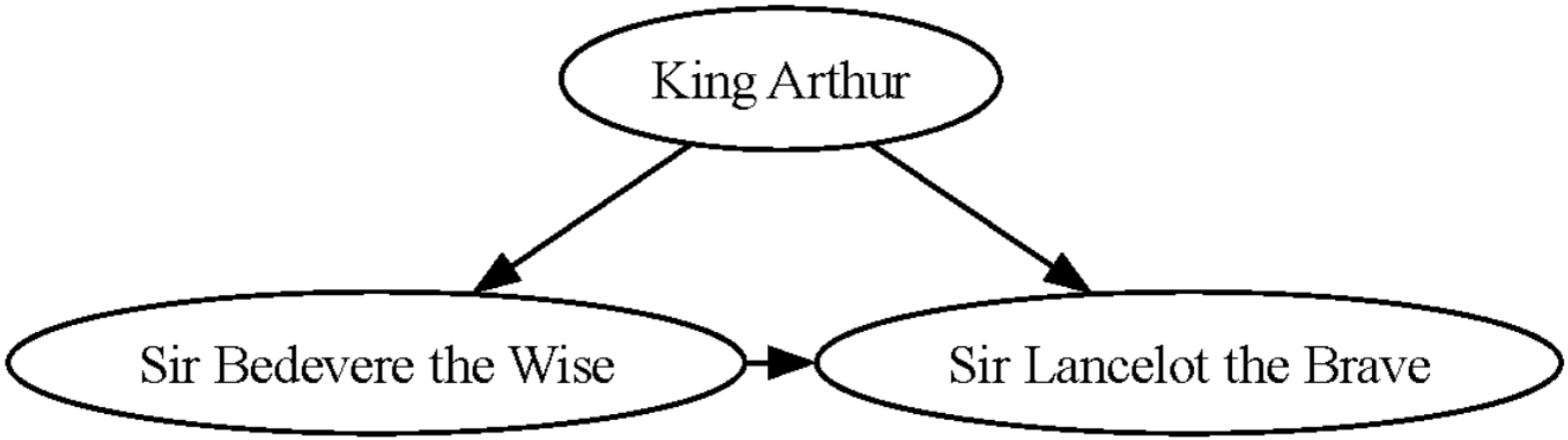
属性付きの
edge_stmt

```
digraph {
    a -> b [...]
}
```

An ***edgeop*** is
-> in directed graphs and
-- in undirected graphs.

DOT言語によるグラフ定義の例

```
digraph "round-table" {
        A [label="King Arthur"]
        B [label="Sir Bedevere the Wise"]
        L [label="Sir Lancelot the Brave"]
        A -> B
        A -> L
        B -> L [constraint=false]
}
```



graphviz
https://graphviz.org/

```
import graphviz

dot = graphviz.Digraph('round-table', comment='The Round Table')

dot.node('A', 'King Arthur')
dot.node('B', 'Sir Bedevere the Wise')
dot.node('L', 'Sir Lancelot the Brave')

dot.edges(['AB', 'AL'])
dot.edge('B', 'L', constraint='false')
```

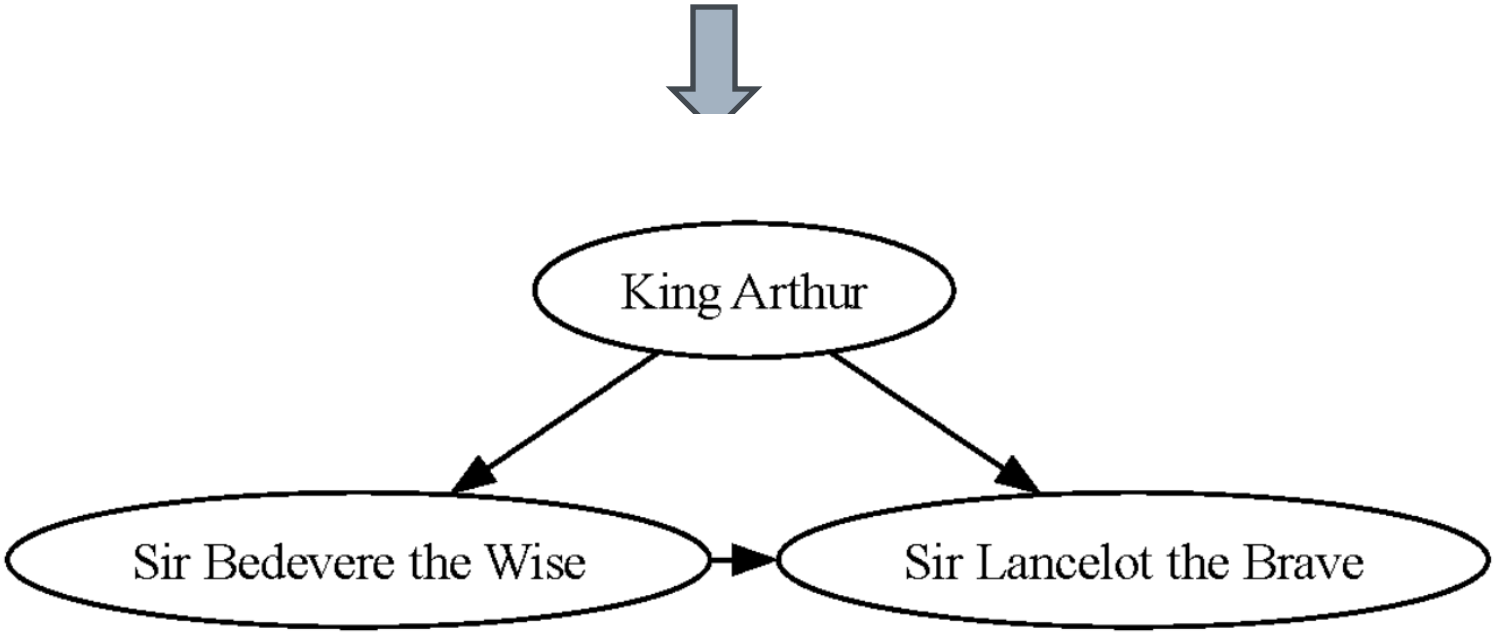
```
print(dot.source)
```

DOT言語でのソースの出力

```
// The Round Table
digraph "round-table" {
        A [label="King Arthur"]
        B [label="Sir Bedevere the Wise"]
        L [label="Sir Lancelot the Brave"]
        A -> B
        A -> L
        B -> L [constraint=false]
}
```

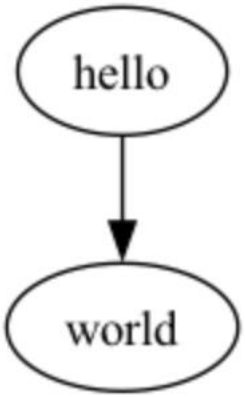
```
dot.render(directory='doctest-output', view=True)
```

DOT言語で記述されたグラフの画像出力



'doctest-output/round-table.gv.pdf'

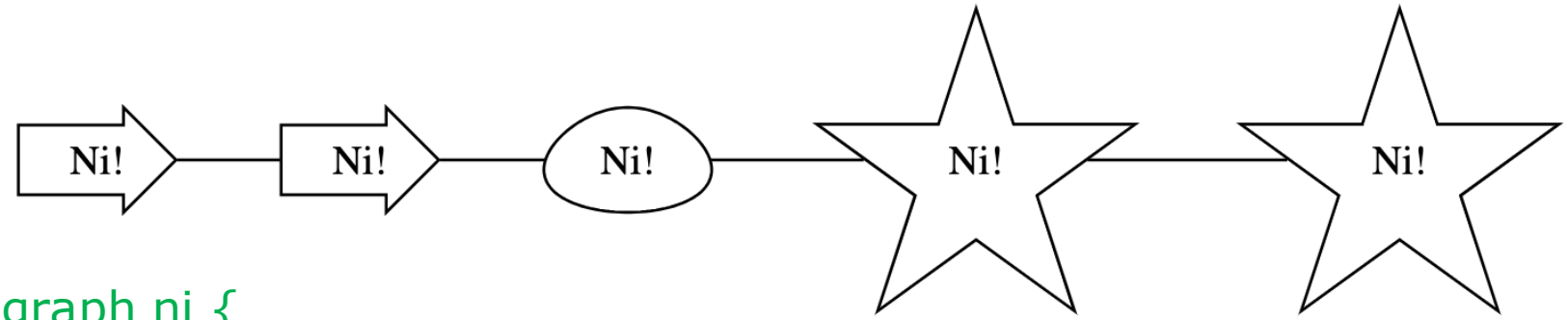
```
import graphviz

dot = graphviz.Digraph('hello')
dot.edge('hello', 'world')
dot.format = 'png'

dot.render(directory='doctest-output' , view=True)
```



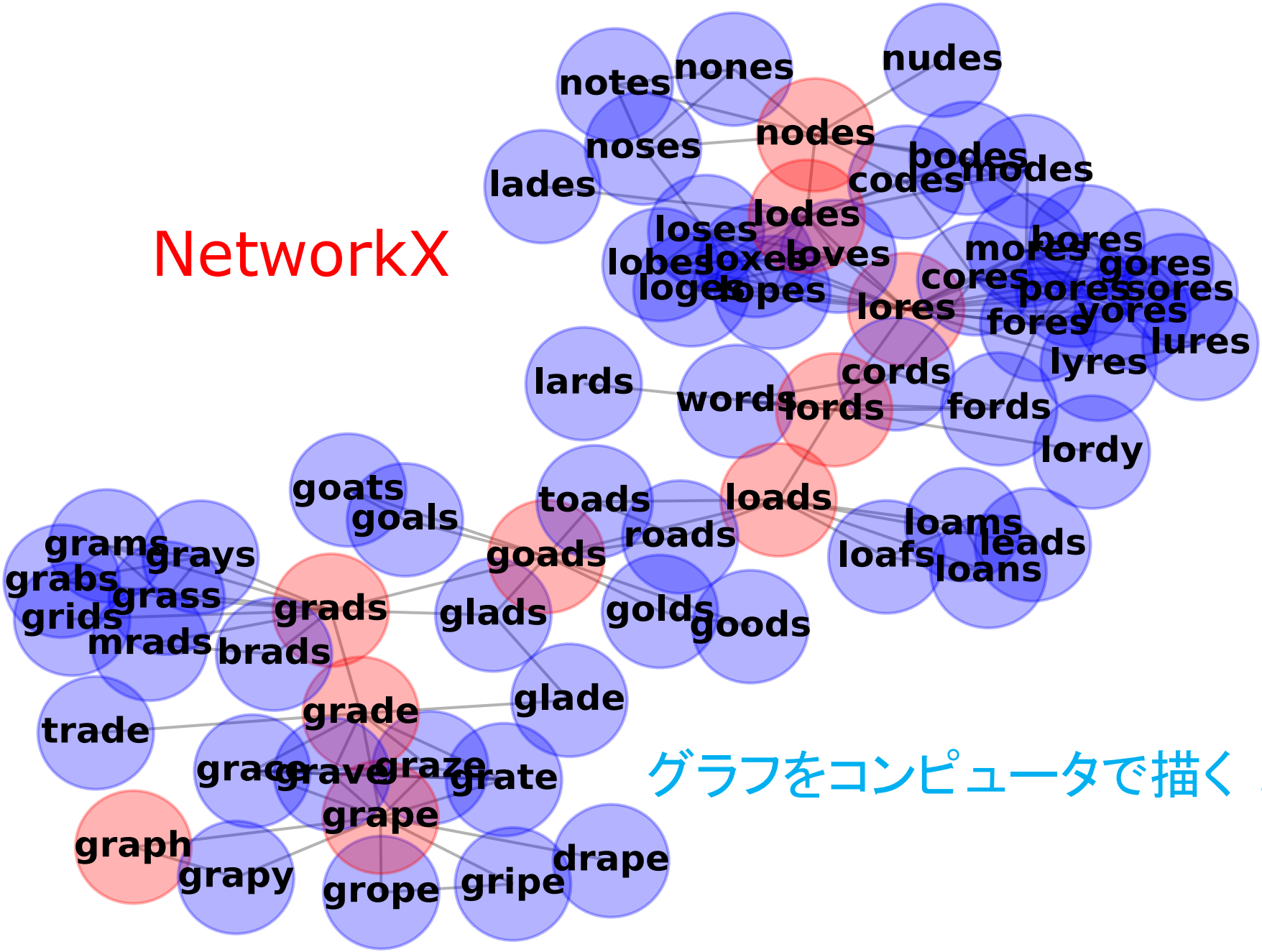
'doctest-output/hello.gv.png'

Attributeの設定

```
import graphviz

ni = graphviz.Graph('ni')

ni.attr('node', shape='rarrow')
ni.node('1', 'Ni!')
ni.node('2', 'Ni!')
ni.node('3', 'Ni!', shape='egg')

ni.attr('node', shape='star')
ni.node('4', 'Ni!')
ni.node('5', 'Ni!’)
ni.attr(rankdir='LR')

ni.edges(['12', '23', '34', '45’])
nl
```




```
graph ni {
        node [shape=rarrow]
        1 [label="Ni!"]
        2 [label="Ni!"]
        3 [label="Ni!" shape=egg]
        node [shape=star]
        4 [label="Ni!"]
        5 [label="Ni!"]
        rankdir=LR
        1 -- 2
        2 -- 3
        3 -- 4
        4 -- 5
}
```



グラフをコンピュータで描く 3

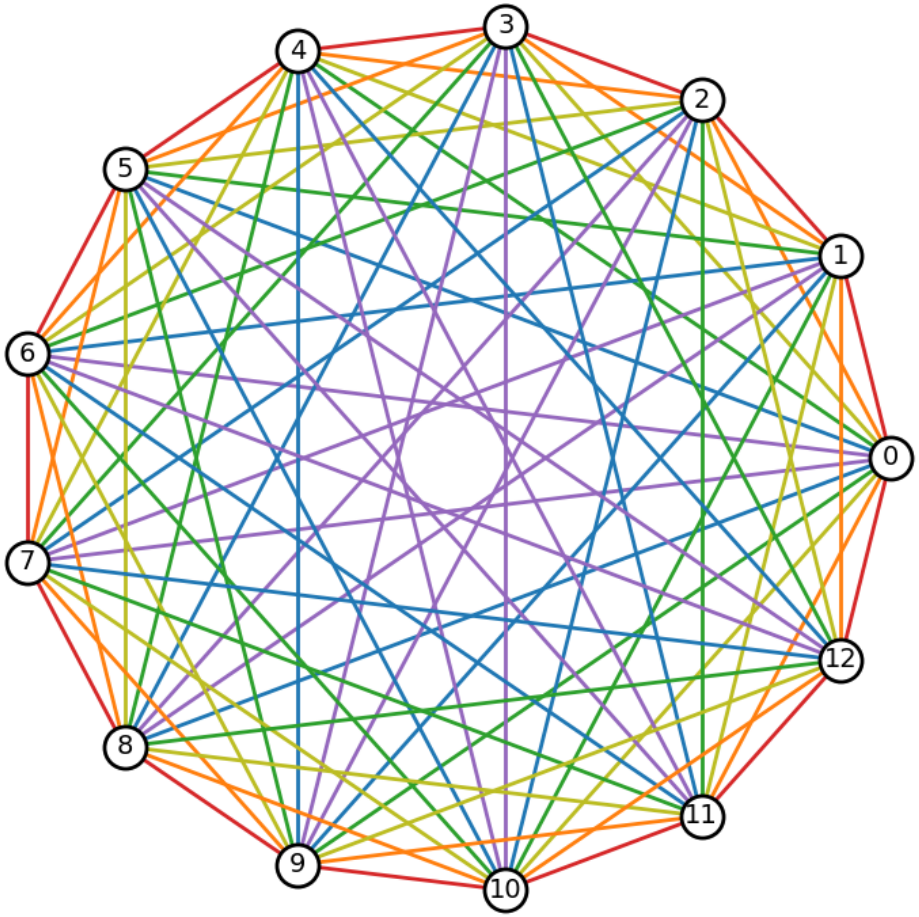
NetworkX

一方のNetworkXは、Pythonの世界ではもっとも成功しているグラフ描画のライブラリーの一つかもしれません。僕は、NetworkX知らなかったのですが、GPT4oに推薦されました。

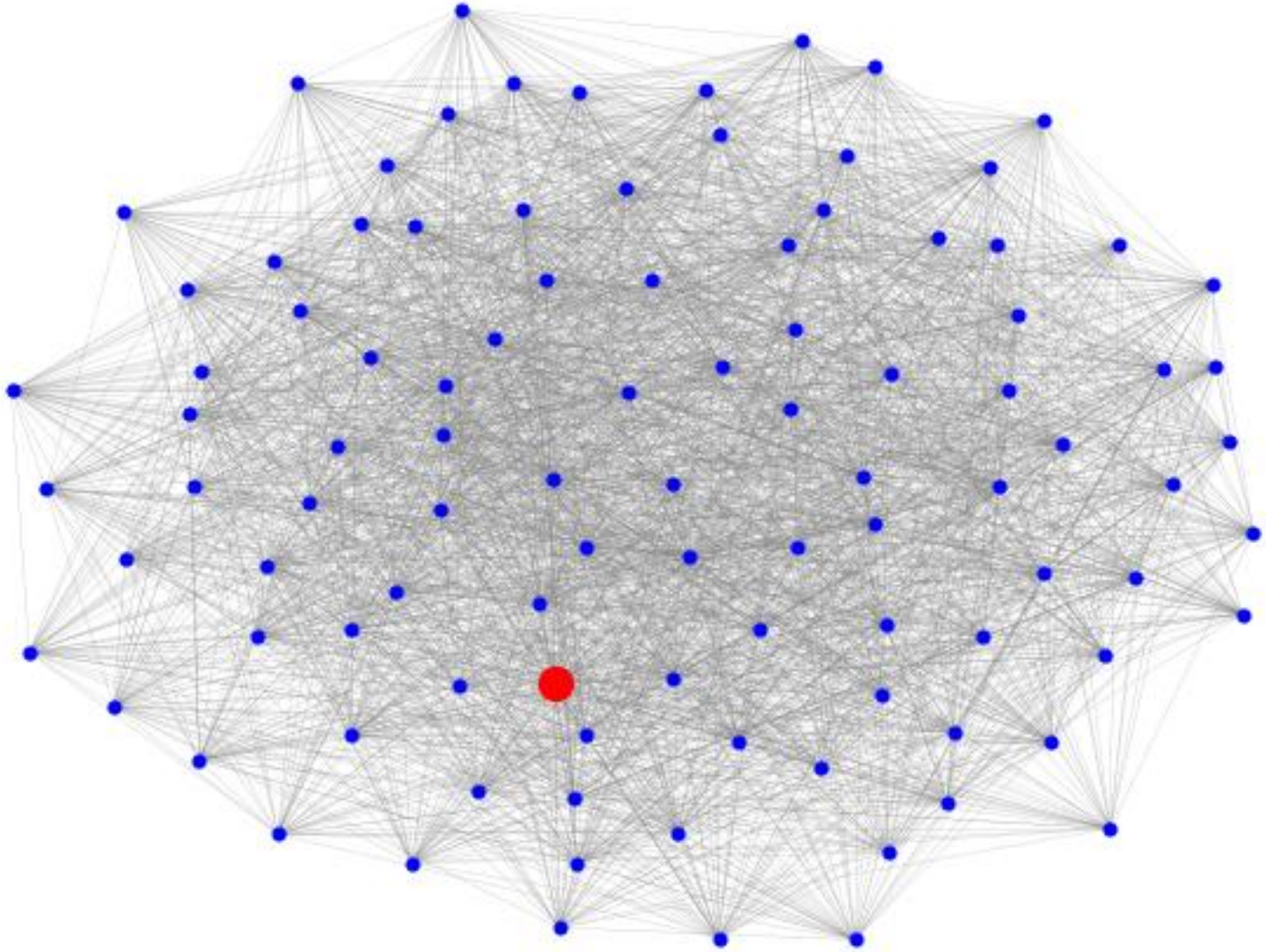
Software for Complex Networks
https://networkx.org/documentation/latest/index.html

NetworkX Tutorial
https://networkx.org/documentation/latest/tutorial.html

NetworkX Examples



https://networkx.org/documentation/latest/auto_examples/index.html



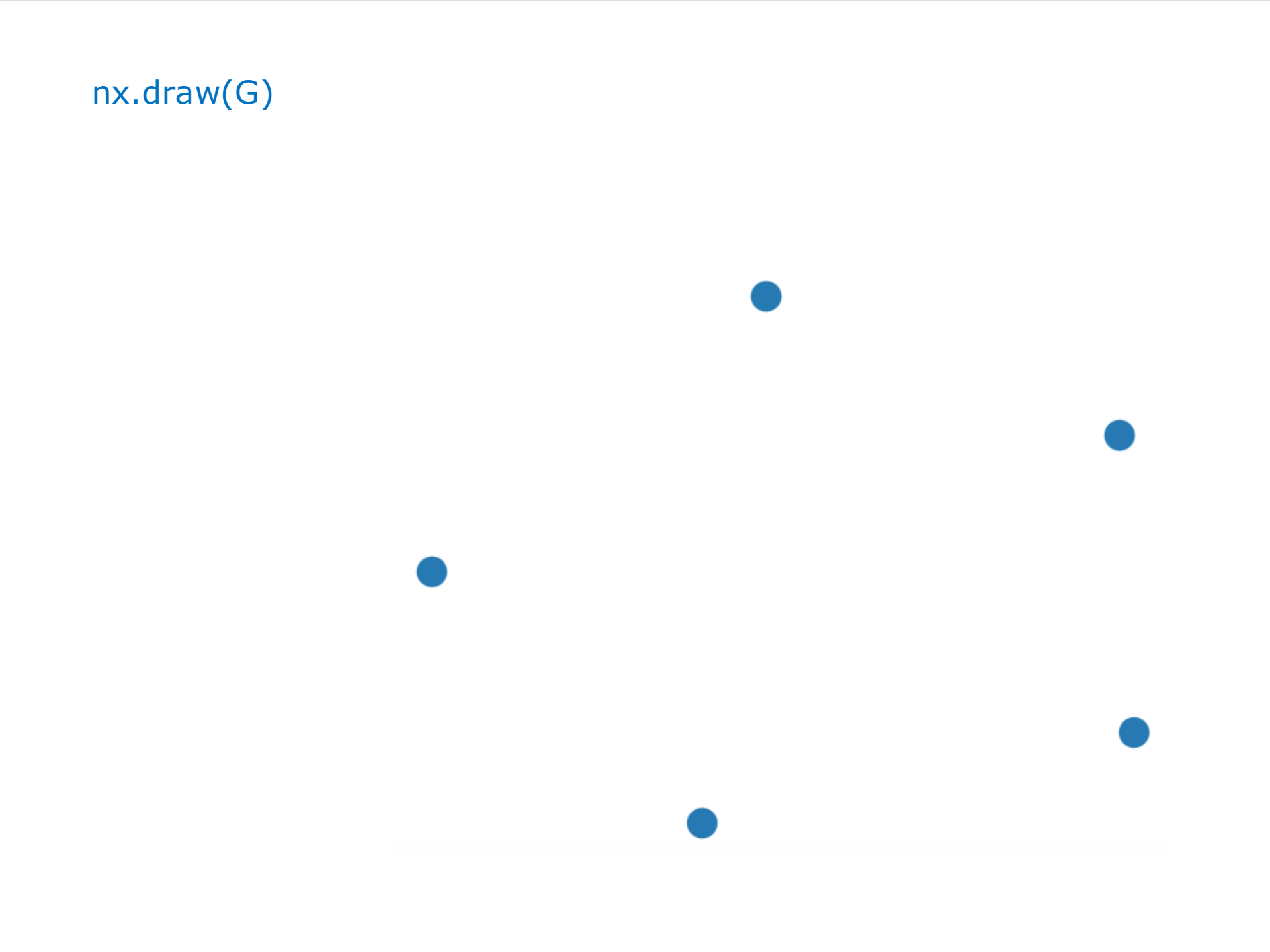
NetworkX
グラフの作成 – nodeから

```
import matplotlib.pyplot as plt
import networkx as nx

G = nx.Graph()

G.add_node(1)
G.add_nodes_from([2, 3])
G.add_nodes_from([(4, {"color": "red"}), (5, {"color": "green"})])

list(G.nodes)
```

グラフの作成 – nodeから

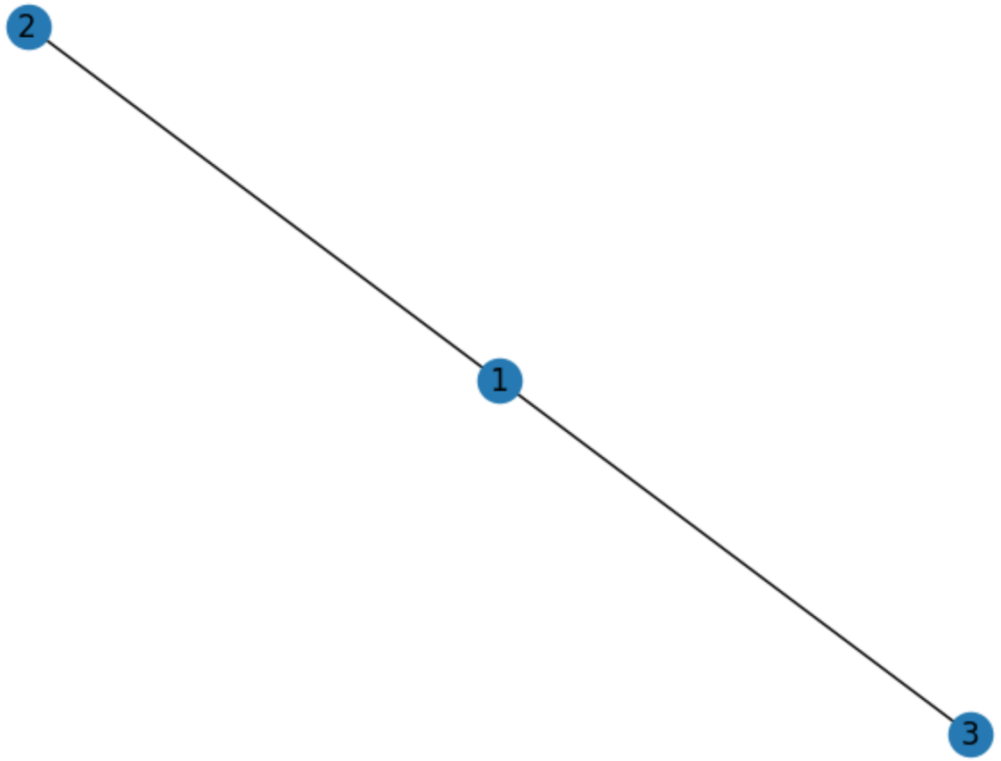
```
import matplotlib.pyplot as plt
import networkx as nx

G = nx.Graph()    // グラフの作成

G.add_node(1)    //  node 1 を追加
G.add_nodes_from([2, 3])    // リスト[2, 3]からnodeを追加
G.add_nodes_from([(4, {"color": "red"}), (5, {"color": "green"})])
                        // color属性を持つnode 4, 5を追加
list(G.nodes)
                        // グラフGのnodeのリストを表示
[1, 2, 3, 4, 5]
```

nx.draw(G)



NetworkX
グラフの作成 – edgeから

```
import matplotlib.pyplot as plt
import networkx as nx

G = nx.Graph()

G.add_edge(1, 2)
G.add_edge(1, 3)

nx.draw(G,with_labels=True)
```



NetworkX
グラフの要素を調べ、操作する

```
[160]: list(G.nodes)
```

```
[160]: [1, 2, 3, 'spam', 's', 'p', 'a', 'm']
```

```
[161]: list(G.edges)
```

```
[161]: [(1, 2), (1, 3), (3, 'm')]
```

```
[162]: list(G.adj[1])  # or list(G.neighbors(1))
```

```
[162]: [2, 3]
```

```
[163]: G.degree[1]  # the number of edges incident to 1
```

```
[163]: 2
```

グラフの要素を消去する

[179]:
```
list(G.nodes)
```

[179]:
```
[1, 2, 3, 'spam', 's', 'p', 'a', 'm']
```

[180]:
```
list(G.edges)
```

[180]:
```
[(1, 2), (1, 3), (3, 'm')]
```

[181]:
```
G.remove_node(2)
G.remove_nodes_from("spam")
list(G.nodes)
```

[181]:
```
[1, 3, 'spam']
```

[182]:
```
list(G.edges)
```

[182]:
```
[(1, 3)]
```

[183]:
```
G.remove_edge(1, 3)
list(G.edges)
```

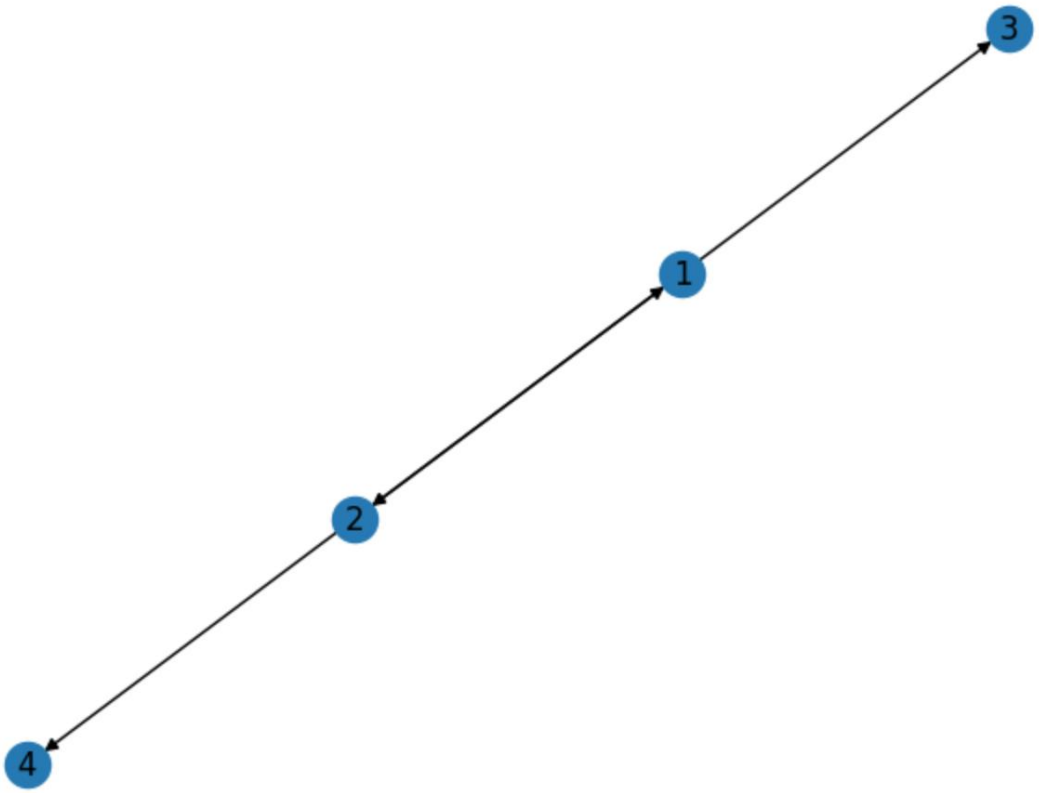
[183]:
```
[]
```

NetworkX
有向グラフの作成

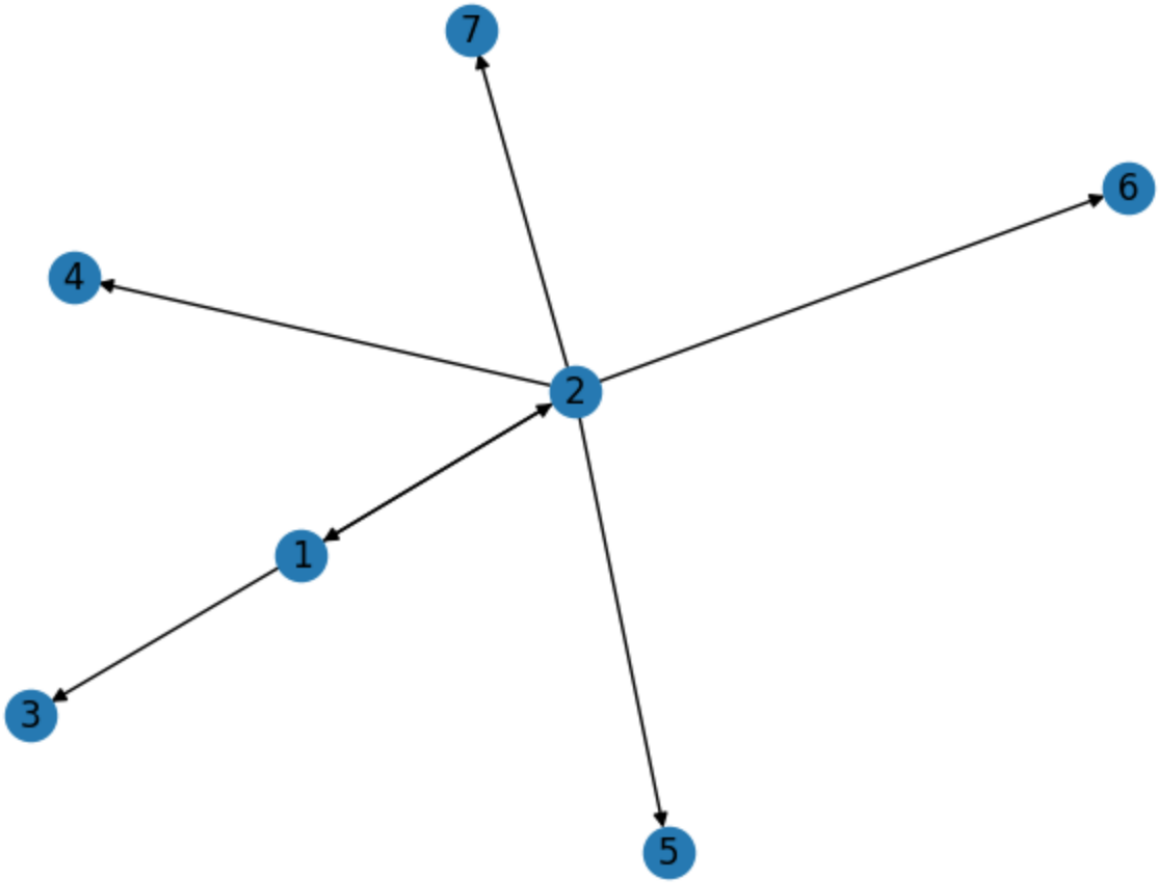
```
import matplotlib.pyplot as plt
import networkx as nx

DG = nx.DiGraph()

DG.add_edge(2, 1)   # adds the nodes in order 2, 1
DG.add_edge(1, 3)
DG.add_edge(2, 4)
DG.add_edge(1, 2)

nx.draw(DG , with_labels=True)
```

有向グラフの作成

```
import matplotlib.pyplot as plt
import networkx as nx

DG = nx.DiGraph()

DG.add_edge(2, 1)   #
DG.add_edge(1, 3)
DG.add_edge(2, 4)
DG.add_edge(1, 2)

nx.draw(DG , with_lab
```



```
DG.add_edge(2,5)
DG.add_edge(2,6)
DG.add_edge(2,7)

nx.draw(DG , with_labels=True)
```



NetworkXとGraphvizの違い

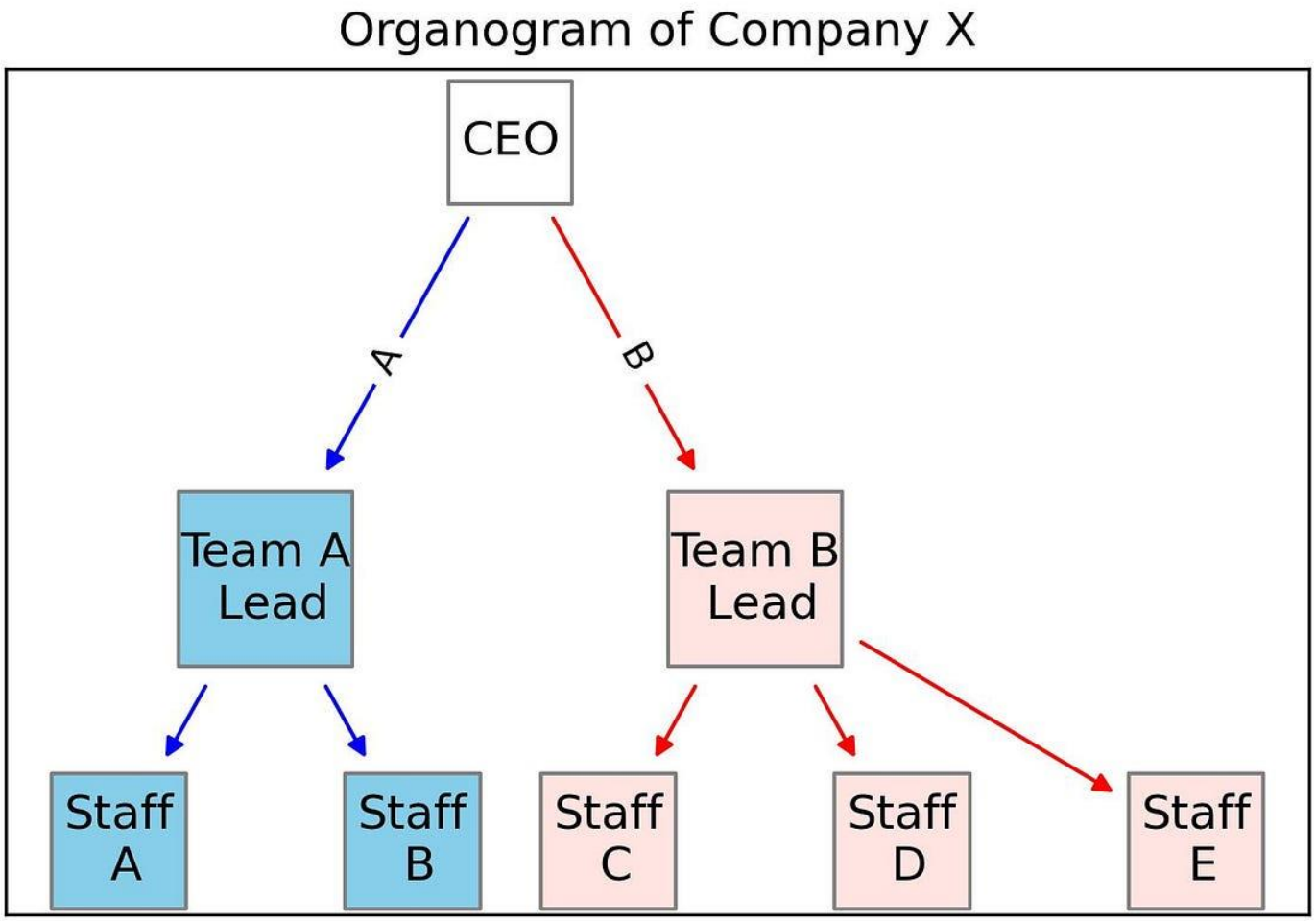
NetworkXとGraphvizでは、表現力に少し違いがあります。

二つを比較した、次の資料を参照してください。

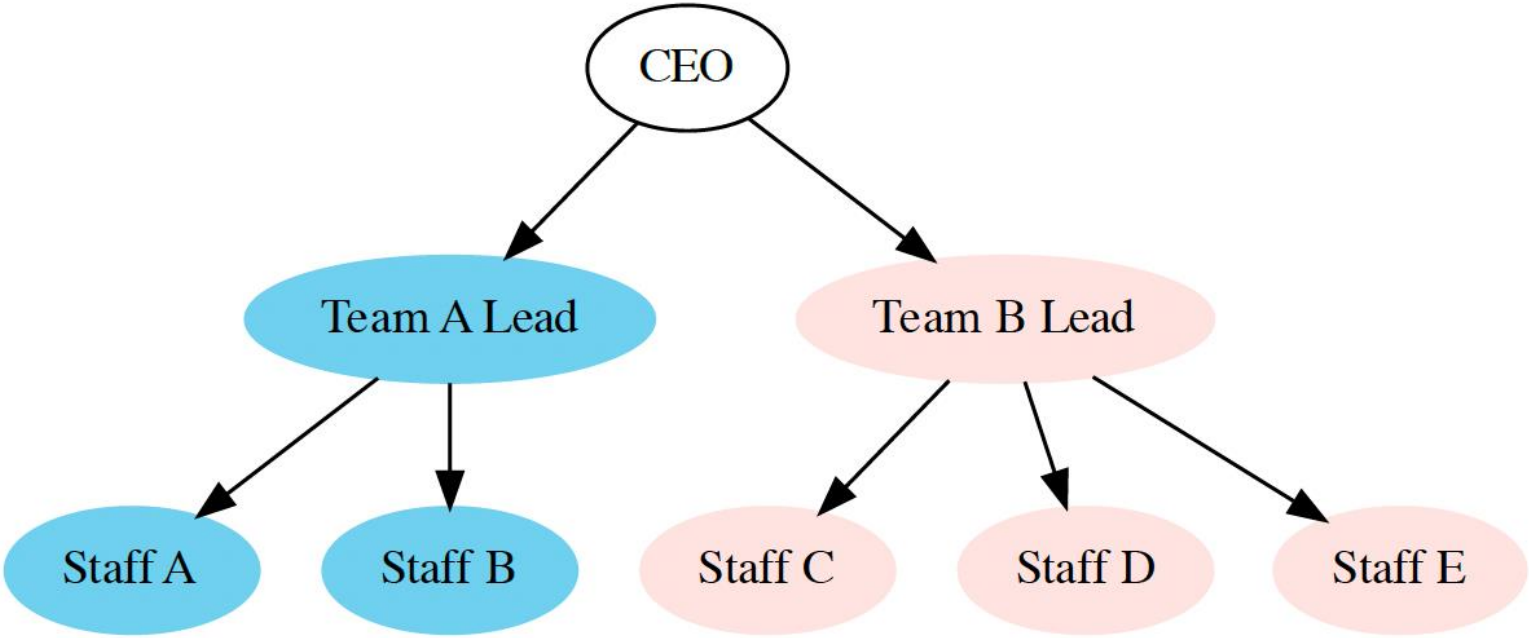
- NetworkX
  https://towardsdatascience.com/graph-visualisation-basics-with-python-part-ii-directed-graph-with-networkx-5c1cd5564daa

- Graphviz
  https://towardsdatascience.com/graph-visualisation-basics-with-python-part-iii-directed-graphs-with-graphviz-50116fb0d670

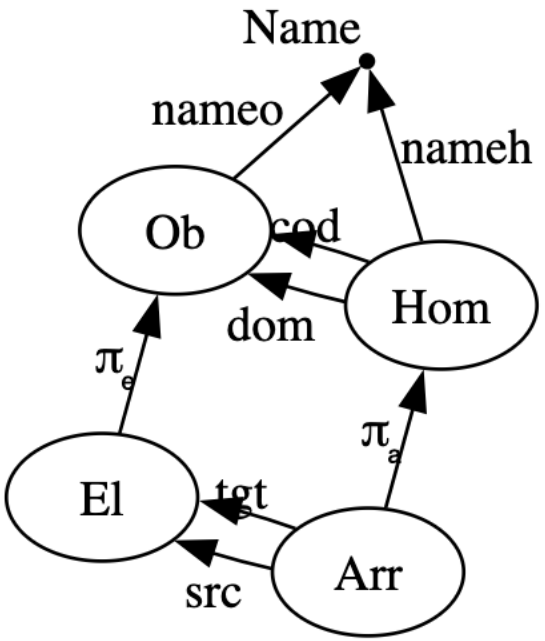
NetworkXで描いた組織図

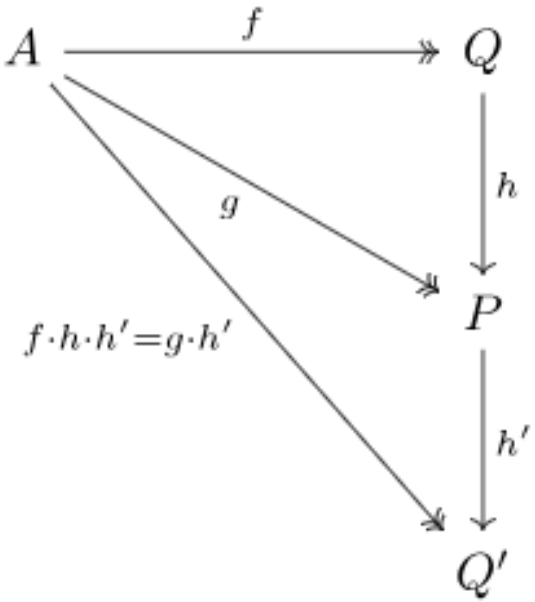


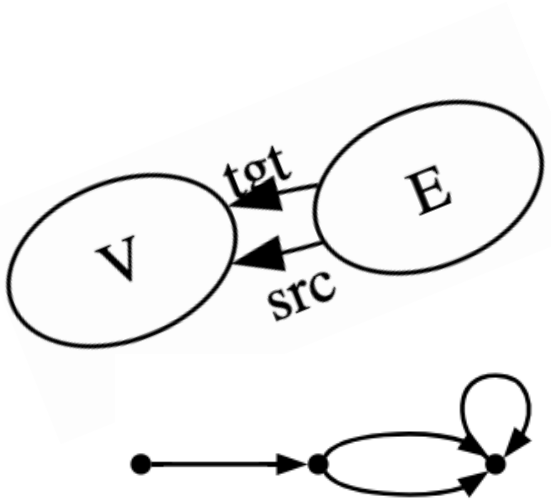
Graphvizで描いた組織図

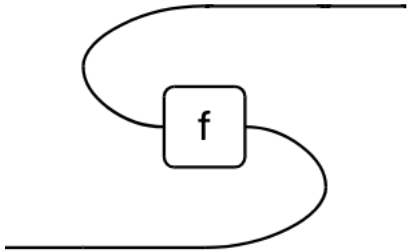


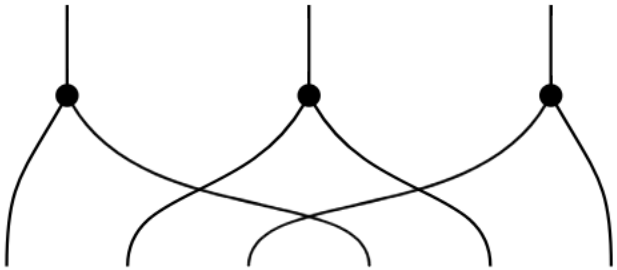
Catlab in AlegebraicJulia



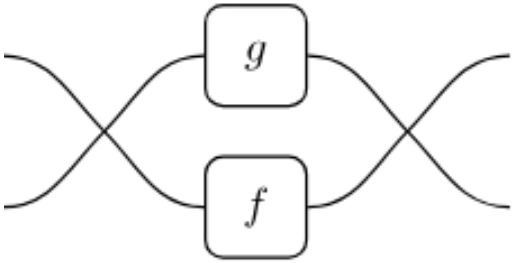






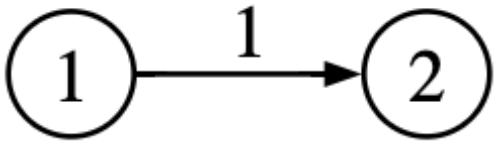


グラフをコンピュータで描く 4

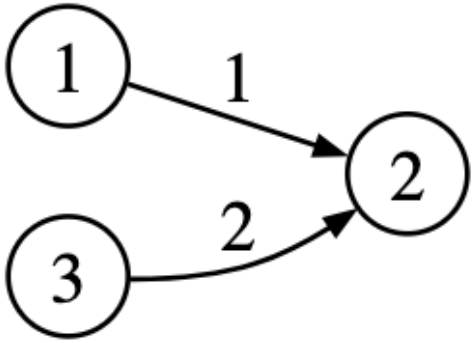


@acset マクロでの定義
C-setとしてのグラフ

```
e = @acset Graph begin
    V = 2
    E = 1
    src = [1]
    tgt = [2]
end

draw(e)
```

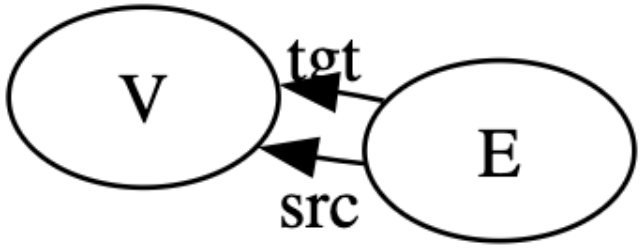


```
w = @acset Graph begin
    V = 3
    E = 2
    src=[1,3]
    tgt=[2,2]
end

draw(w)
```

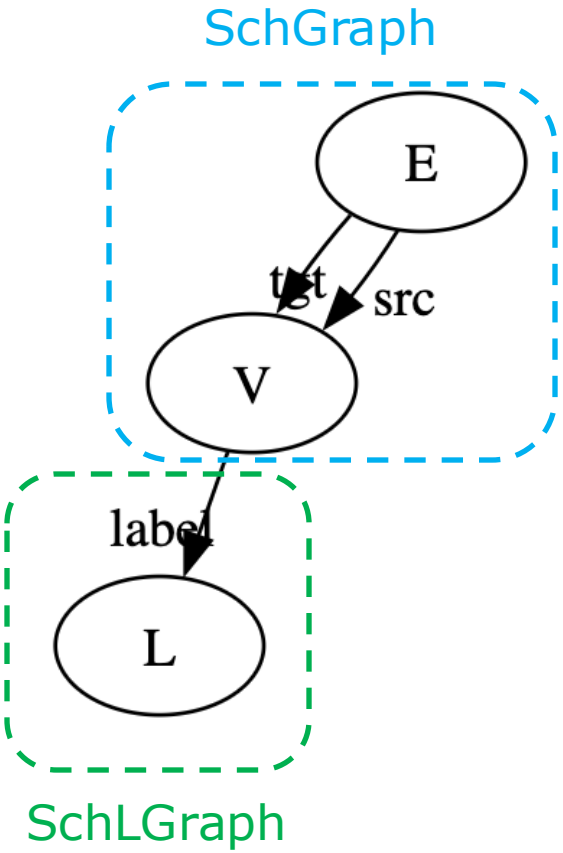


@present マクロでの定義
グラフのスキーマとしてのグラフ

```
using Catlab.Theories, Catlab.CategoricalAlgebra
using Catlab.Graphs, Catlab.Graphics
using Colors

@present SchGraph(FreeSchema) begin
  V::Ob
  E::Ob
  src::Hom(E,V)
  tgt::Hom(E,V)
end

to_graphviz(SchGraph)
```



```
@present SchLGraph <: SchGraph begin
  L::Ob
  label::Hom(V,L)
end

to_graphviz(SchLGraph)
```



Wiring Diagram (= String Diagram) としてのグラフ

```
using Catlab.WiringDiagrams
using Catlab.Graphics
import Catlab.Graphics: Graphviz

show_diagram(d::WiringDiagram) = to_graphviz(d,
  orientation=LeftToRight,
  labels=true, label_attr=:xlabel,
  node_attrs=Graphviz.Attributes(
    :fontname => "Courier",
  ),
  edge_attrs=Graphviz.Attributes(
    :fontname => "Courier",
  )
)
```

カテゴリー論へのインターフェース

```
using Catlab.Theories

A, B, C, D = Ob(FreeBiproductCategory, :A, :B, :C, :D)
f = Hom(:f, A, B)
g = Hom(:g, B, C)
h = Hom(:h, C, D)

f
```

$$f : A \to B$$

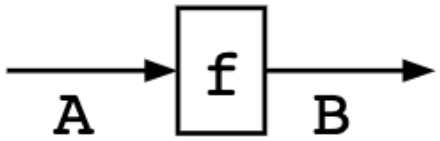
Generator

```
f, g, h = to_wiring_diagram(f), to_wiring_diagram(g),
to_wiring_diagram(h)
```

f

```
WiringDiagram{Catlab.Theories.ThBiproductCategory.Meta.T}(
[:A], [:B],
[ -2 => {inputs},
  -1 => {outputs},
   1 => Box(:f, [:A], [:B]) ],
[ Wire((-2,1) => (1,1)),
  Wire((1,1) => (-1,1)) ])
```

```
show_diagram(f)
```

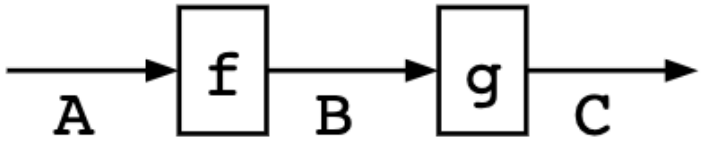


Composition

compose(f,g)

```
WiringDiagram{Catlab.Theories.ThBiproductCategory.Meta.T}(
 [:A], [:C],
 [ -2 => {inputs},
   -1 => {outputs},
    1 => Box(:f, [:A], [:B]),
    2 => Box(:g, [:B], [:C]) ],
 [ Wire((-2,1) => (1,1)),
   Wire((1,1) => (2,1)),
   Wire((2,1) => (-1,1)) ])
```

show_diagram(compose(f,g))

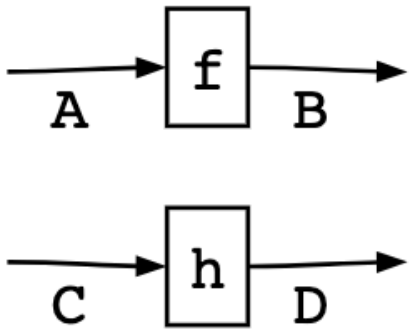


Monoidal Product

otimes(f,h)

```
WiringDiagram{Catlab.Theories.ThBiproductCategory.Meta.T}(
[:A,:C], [:B,:D],
[ -2 => {inputs},
  -1 => {outputs},
   1 => Box(:f, [:A], [:B]),
   2 => Box(:h, [:C], [:D]) ],
[ Wire((-2,1) => (1,1)),
  Wire((-2,2) => (2,1)),
  Wire((1,1) => (-1,1)),
  Wire((2,1) => (-1,2)) ])
```

show_diagram(otimes(f,h))

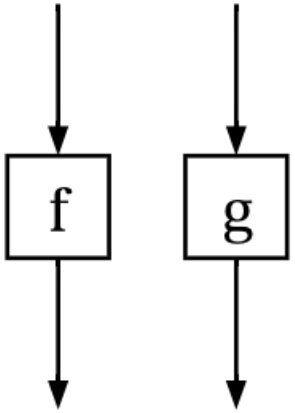


Drawing wiring diagrams in Graphviz

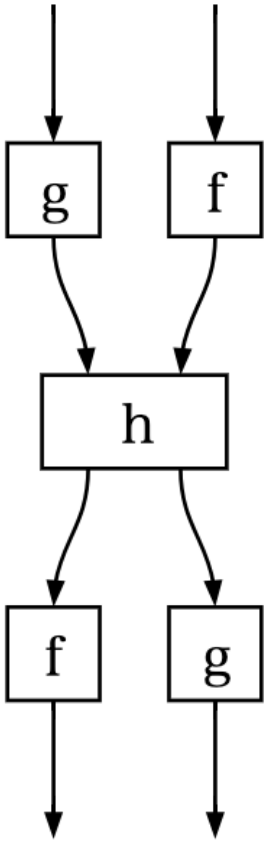
```
using Catlab.Theories

A, B = Ob(FreeSymmetricMonoidalCategory, :A, :B)
f = Hom(:f, A, B)
g = Hom(:g, B, A)
h = Hom(:h, otimes(A,B), otimes(A,B));

to_graphviz(f)
```

to_graphviz(compose(f,g))

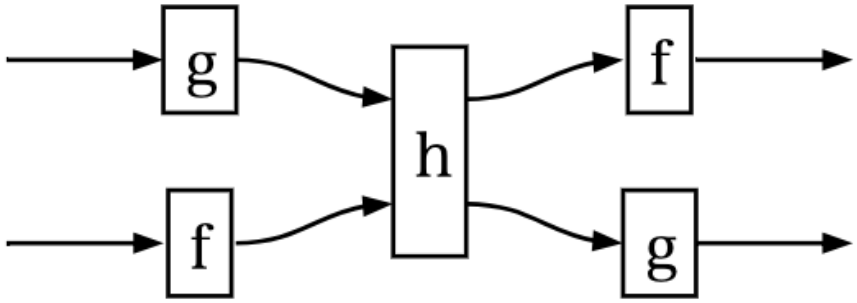
to_graphviz(otimes(f,g))



```
composite = compose(otimes(g,f), h, otimes(f,g))
to_graphviz(composite)
```



to_graphviz(composite, orientation=LeftToRight)



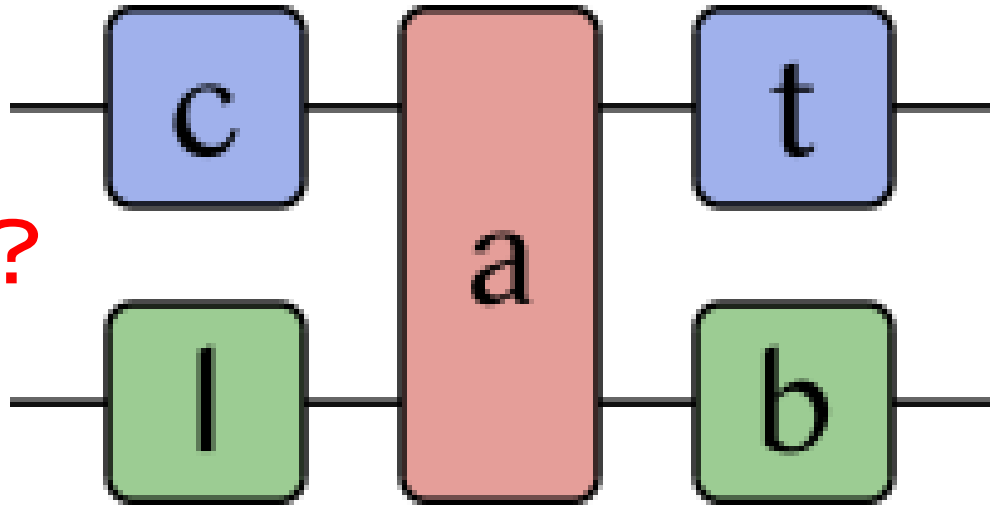
Part 3

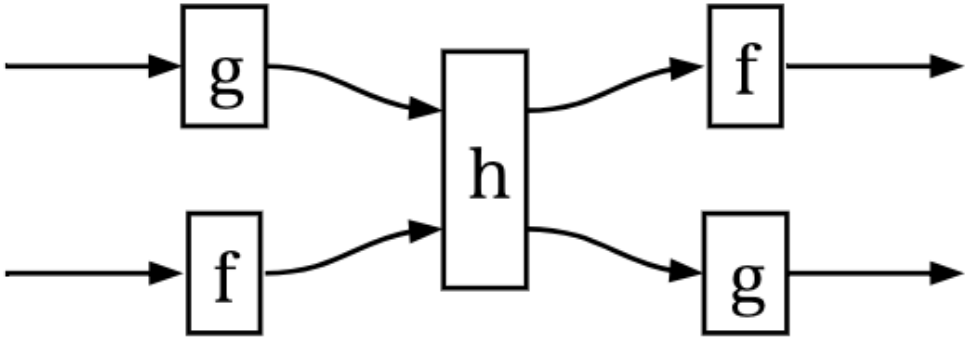
グラフのカテゴリー論的特徴づけ　C-set

Agenda Part 3
グラフのカテゴリー論的特徴づけ C-set

- Catlab とは何か？
- C-Setとはなにか？
  - グラフをFunctorで捉える
  - CatlabでのC-setプログラミング
  - 無向グラフと反射的グラフのスキーマ

Catlab とは何か？





応用カテゴリー論の
ツールとしてのCatlab

```
composite = compose(otimes(g,f), h, otimes(f,g))
```

Catlab は
カテゴリー論の応用のためツール

Catlab.jlはJulia言語で書かれた、応用的で計算的なカテゴリー理論のためのフレームワークです。

Frame work for

- Applied Category Theory
- Computational Category Theory

Catlabは、科学・技術分野へのカテゴリー理論の応用のためのプログラミング・ライブラリーと対話的インターフェースを提供します。（単なるグラフ・ライブラリーではありません。）

Provides

- Programming Library
- Interactive Interface

Catlab.jlは、一般化された代数理論 (Generalized Algebraic Theories) として形式化可能なカテゴリー構造であれば、どのようなものでもサポートできます。

こうしたCatlabの能力は、AlgebraicJuliaのもう一つの基本的パッケージであるGatlabによって与えられています。

Gatlabについては、セミナーの次の章で紹介したいと思います。

https://github.com/AlgebraicJulia/GATlab.jl

Catlabとは何か？

Catlabとは、以下のようなものです。（あるいは、そういうものになるでしょう。）

第一に、それはプログラミング・ライブラリです。
第二に、それはインタラクティブな計算環境を提供します。
第三に、それはコンピュータ上の代数システムとして機能します。

プログラミング・ライブラリとしてのCatlab

- Catlabは応用カテゴリー理論のためのデータ構造、アルゴリズム、およびSerializationを提供します。

- マクロは、カテゴリー論的な基本的な方向性と型安全な記号操作システムの仕様を指示するための便利な構文を提供します。

- Wiring Diagram(配線図)は、別名ストリング・ダイアグラムとも呼ばれ、特殊なデータ構造によってサポートされます。Wiring diagramは、GraphML(XMLベースのフォーマット)やJSONとの間でシリアライズすることができます。

インタラクティブな計算環境としてのCatlab

Catlabは、Jupyterノートブックでインタラクティブに使用することもできます。

記号式はLaTeXを使って表示され、Wiring DiagramはCompose.jl、Graphviz、TikZを使って可視化されます。

コンピュータ代数システムとしてのCatlab

Catlabはカテゴリー代数のコンピュータ代数システムとして機能します。

ほとんどのコンピュータ代数システムとは異なり、すべての式は一般化代数理論(Generalized Algebraic Theories -- これが先に見たGATです)と呼ばれる従属型理論の断片を用いて型付けされます。

我々は、カテゴリーや対称モノイダルカテゴリーのようないくつかの重要なdoctrinに関して、word problemを解き、式を正規形に還元するためのコアアルゴリズムを実装しています。

古典的抽象代数の計算機代数については、AbstractAlgebra.jlとNemo.jlを参照してください。

Catlabがそうではないもの

Catlabは現在のところ、以下のようなものではありません。

しかし、いずれこれらの方向に進化する可能性を否定するものではありません。

- 自動定理証明機: コンピュータ代数と自動定理証明の間には重なる部分があるのですが、Catlabは正しさの形式的な証明書（いわゆる「証明」）を作成しないため、定理証明機と考えることはできません。

- 証明アシスタント: 同様に、Catlabは形式的に検証可能な証明を生成しないため、証明アシスタントではありません。形式的なな検証はプロジェクトの範囲外です。

- グラフィカル・ユーザー・インターフェース: CatlabはWiring Diagramエディタやその他のグラフィカル・ユーザー・インターフェースを提供しません。

これはCatlabが主にプログラミング・ライブラリであり、ユーザー向けのアプリケーションではないからです。

しかし、AlgebraicJuliaのエコシステムには、Wiring Diagramやペトリネットなどを操作するためのグラフィカル・ユーザー・インターフェースを提供する別のプロジェクトSemagrams.jlがあります。

スキーマ・カテゴリーC

$$E \underset{\text{tgt}}{\overset{\text{src}}{\rightrightarrows}} V$$

グラフをFunctorで捉える

Functor F

$F(E) =$ $\{1,2,\ldots,m\}$

$F(C)$のグラフの辺の集合(m個)

$F(V) =$ $\{1,2,\ldots,n\}$

$F(C)$のグラフの頂点の集合(n個)

C-Set カテゴリー F(C)

C-set とは何か 1

グラフをFunctorで捉える

このセッションでは、Catlabではグラフをどのように捉えているのかを紹介します。それは、とてもユニークなものです。

一口で言えば、それは、グラフをfunctorとして捉えるというものです。

このユニークなアイデアは強力なものです。ただ、最初はその意味が分かりにくいと思います。

このセッションでは、functorの性質を確認しながら、Catlabでの実装の具体的な例を通じて、この考え方に慣れることを目標にしたいと思います。

グラフは、C-set

この「グラフをfunctorとして捉える」 という考え方を、数学的には「グラフはC-setである」と言います。この考え方を、まず説明します。

カテゴリーCから集合のカテゴリー Set へのfunctor $F: C \to Set$ が与えられた時、Cから あるfunctor Fの作用を受けて生み出される集合のことを C-setと呼びます。

「グラフはC-setである」ということは、カテゴリーCと $F: C \to Set$ であるfunctor F が与えられた時、グラフはF(C)で表される構造を持った集合であるということです。

あまりに、抽象的...

こうした定義は、あまりに抽象的なものにみえます。

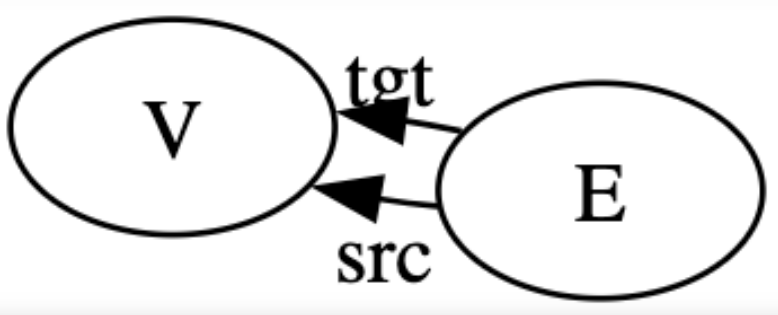
こうした定義で、多様なグラフを表現することができるのでしょうか？ そうした疑問は当然です。

ただ、「グラフは C-set である」という主張を、この命題でのカテゴリーCはどういうもので、functor $F: C \to Set$がどのようなものかを具体的に検討していくと、驚くべきことが判明します。

あまりに、単純 ...

最初の驚きは、「グラフは C-set である」というときのカテゴリー C の構造が驚くほど単純であるということです。

functor Fの働きで、全ての「有向グラフ」を生成するカテゴリー Cは、次の形をしています。



この抽象的で単純なカテゴリーCを、「グラフのスキーマ」と呼びます。

この抽象的で単純なカテゴリーC を、「グラフのスキーマ(Schema)」と呼びます。

「グラフは C-set である」という時、カテゴリー C は、「グラフのスキーマ」です。

抽象的な「グラフのスキーマ」Cに、あるfunctor Fが作用して生み出される具体的な集合のカテゴリーが、グラフを表現するC-setだと言うことなのです。

それは少し、抽象的な議論に感じられるでしょう。当面は、もう少し具体的にCというカテゴリーにについて考えてみましょう。

全ての「有向グラフ」を生成するカテゴリー C

カテゴリー C

$$E \underset{\text{tgt}}{\overset{\text{src}}{\rightrightarrows}} V$$

全ての「有向グラフ」を生成するカテゴリー C は、次のものからなります。

- Cの二つのオブジェクト: $E, V \in Obj(C)$
  - EはEdgeでグラフの辺の意味
  - VはVertexでグラフの頂点の意味
- Cの二つの射: $src, tgt : E \to V \in Hom(C)$
  src, tgt は、辺が与えられたとき頂点を返す
  - srcはsourceで辺の始点
  - tgtはtargetで辺の終点

カテゴリー C に、functor F を適用した カテゴリー F(C)はどういうものになるのか？

functor $F: C \to Set$ は、Cのオブジェクトを F(C)のオブジェクトに、Cの射を F(C)の射に移します。

ですので、結果として得られるカテゴリー F(C) (これを C-setと呼びます)は、次のものからなります。

- F(C)の二つのオブジェクト: $F(E), F(V) \in Obj(F(C))$
- F(C)の二つの射: $\mathrm{F}(src), F(tgt): F(E) \to F(V) \in Hom(F(C))$

注意して欲しいのは、$F(E), F(V) \in Set$ であり、$\mathrm{F}(src), F(tgt): F(E) \to F(V)$は$Set$上の関数であるということです。

抽象的なCから、具体的な F(C)を考える
スキーマからC-setへ

このように、カテゴリーCにfunctor Fを適用した F(C) すなわち C-setは、集合の上に定義された具体的なものになります。

最初に見た スキーマのカテゴリーCは驚くほど単純なものでしたが、それは抽象的なものでした。

スキーマのカテゴリーCが表現していることは、「グラフには頂点と辺があること $E, V \in Obj(C)$」、「辺は始点となる頂点と終点となる頂点を持つ $src, tgt : E \to V \in Hom(C)$ 」ということだけでした。

例えば、あるグラフが、n個の頂点を持つというような具体的な性質を、このスキーマ・カテゴリーCは表現できません。

F(V)はグラフの具体的な頂点の集合を表す

ただ、グラフがn個の頂点を持つことを、functor Fを使うと、次のように表現できます。

$$F(V) = \{1,2, \dots, n\}$$

(i番目の頂点に、数字iを割り当てようということです。)

抽象的なスキーマ・カテゴリーCのオブジェクトVに、functor Fを適用したF(V)は、カテゴリーF(C)の中では、グラフの頂点の集合を具体的に(数字で)表現したものになります。

F(E)はグラフの具体的な辺の集合を表す

それでは、カテゴリー F(C)のもう一つのオブジェクトであるF(E)についてはどうでしょう？

先のグラフの頂点への数字の割り当てと同様に、F(E)の要素にも、一つの数字を割り当てることにしましょう。

$$F(E) = \{1,2, \dots, m\}$$

(i番目の辺に、数字iを割り当てようということです。)

この割り当ては、グラフがm個の辺を持っていることを表します。

注意して欲しいのは、グラフの頂点の個数nと辺の個数mは独立なので、

$$F(V) = \{1,2, \dots, n\}$$
$$F(E) = \{1,2, \dots, m\}$$

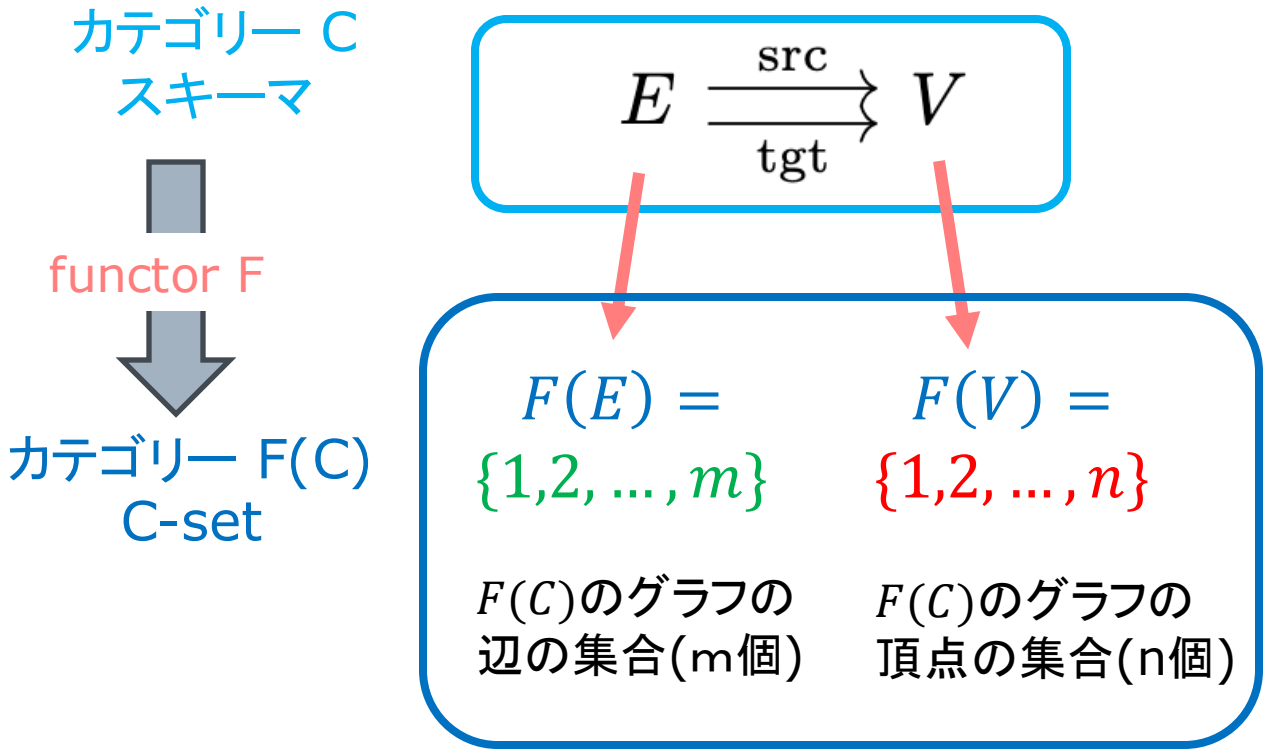
だとしても、$n = m$とは限りません。

さらに言うと、$1 \in F(V)$で$1 \in F(E)$だという時、同じ数字1は異なる対象を表しています。

前者の1は1番目の頂点を、後者の1は1番目の辺を表しています。

Cのオブジェクトと F(C)のオブジェクトの関係

いったん ここまでの、CのオブジェクトE, VとF(C)のオブジェクト$F(E), F(V)$の関係を整理しておきましょう。



F(C)の射の働き

残念ながら、F(C)のオブジェクト$F(E), F(V)$を見ているだけでは、グラフの頂点の数と辺の数が分かるだけで、それ以上具体的にグラフの形がわかるわけではありません。

グラフの形、ある頂点がどの辺で結ばれているかを知るためには、Cの二つの射 srcとtgtがfunctor Fで移されたF(C)の射であるF(src)とF(tgt)の働きが必要になります。

スキーマ・カテゴリーCでは、

$$src : E \to V$$

$$tgt : E \to V$$

でしたので、

C-setカテゴリーであるF(C)では

$$F(src): F(E) \to F(V)$$

$$F(tgt): F(E) \to F(V)$$

となることはわかります。

先に見た頂点・辺への数字の割り当てを利用すれば、

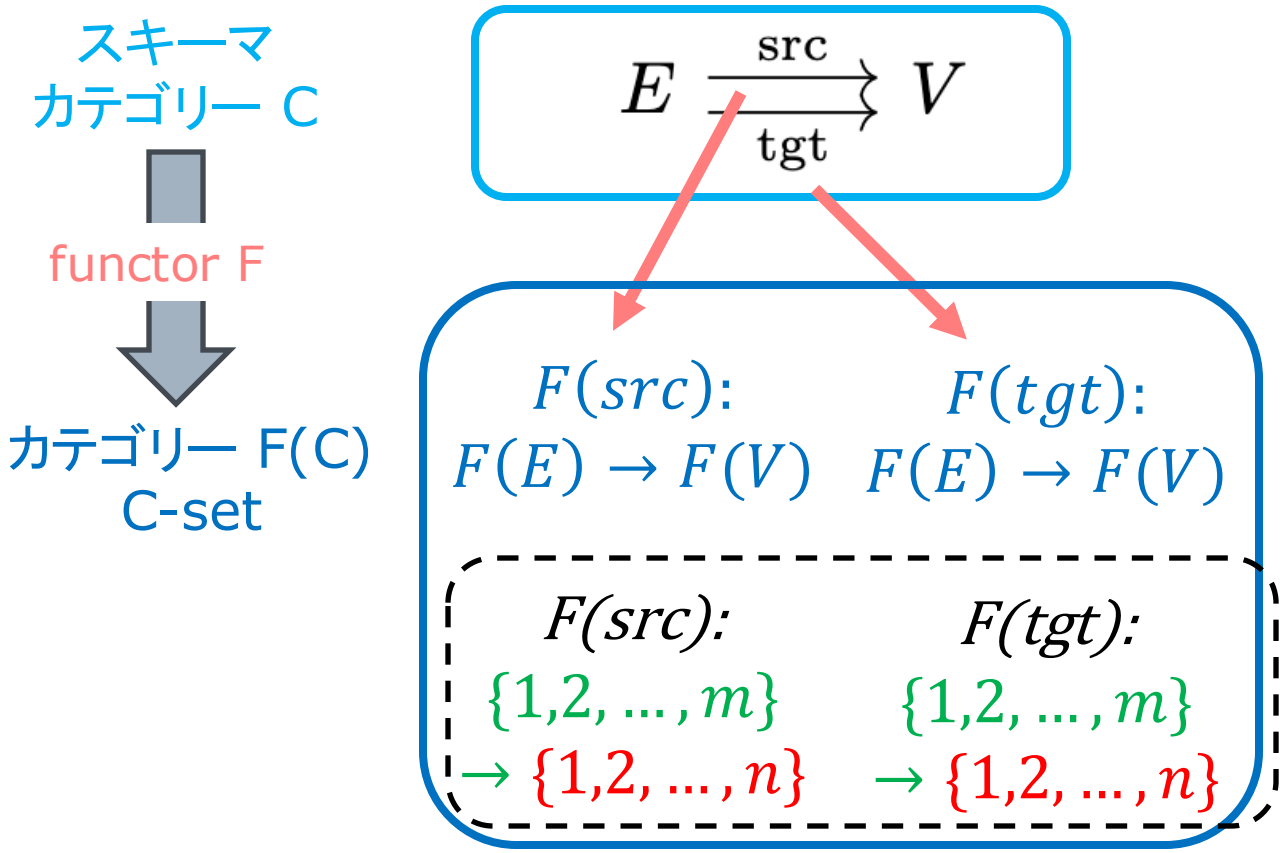
$$F(src) : \{1,2, \dots, m\} \to \{1,2, \dots, n\}$$

$$F(tgt) : \{1,2, \dots, m\} \to \{1,2, \dots, n\}$$

このように、F(C)上の二つの射F(src),F(tgt)は、いずれも辺の集合 $\{1,2, \dots, m\}$ 上で定義され、頂点の集合$\{1,2, \dots, n\}$ に値を取る関数です。

Cの射とF(C)の射の関係

Cの射 src, tgt とF(C)の射$F(src), F(tgt)$の関係。



関数$F(src)$, $F(tgt)$ の定義を通じて 具体的にfunctor F を構成する

$$F(src):\{1,2,\dots,m\} \to \{1,2,\dots,n\}$$
$$F(tgt):\{1,2,\dots,m\} \to \{1,2,\dots,n\}$$

という条件は、Fがfunctorであることから自然に導かれた性質なのですが、この条件だけからは、具体的に関数$F(src)$, $F(tgt)$の形を決めることはできません。

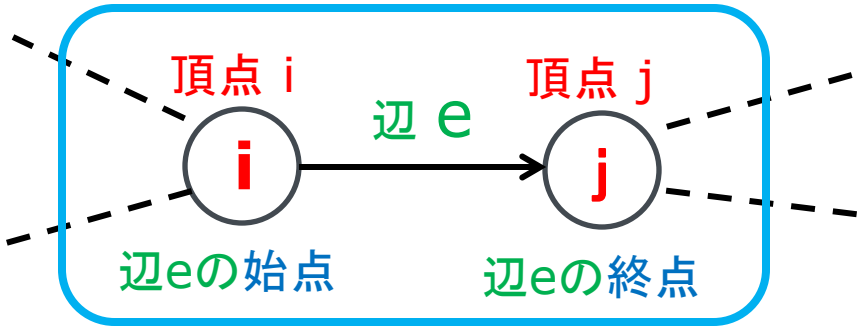
同じ定義域(domain) $\{1,2,\dots,m\}$と、同じ値域(codomain) $\{1,2,\dots,n\}$を持つ自然数上の関数は、無数に存在しますから。

でも、そのことは、逆に、functor Fを与えるときに、定義域と値域の条件の枠の中で(その条件は、Fがfunctorであることから自動的に保証されます)、自由に関数$F(src)$, $F(tgt)$を定義していいことを意味します。

グラフの形と、関数$F(src)$, $F(tgt)$ の定義

ここでは、グラフの形と関数$F(src)$, $F(tgt)$ の定義を結びつけて論じます。

グラフの中に、辺eと頂点i, j を含む次のような部分グラフがあったとします。



このとき、

$$\boldsymbol{F(src)(e) = i}$$
$$\boldsymbol{F(tgt)(e) = j}$$

と、定義します。

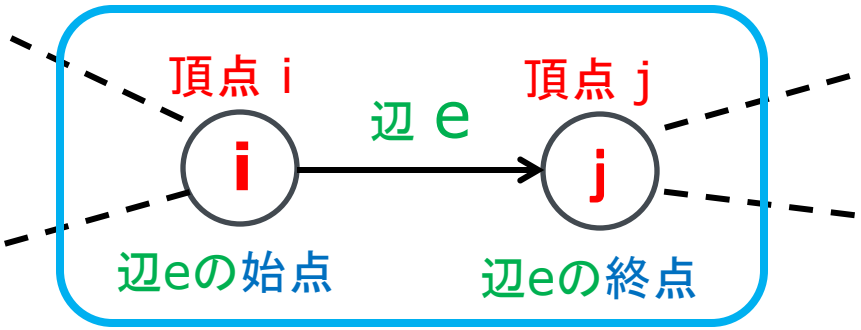
このように、一つの辺eを含む部分グラフに注目すると、それまで定義域と値域の定義しか持たなかった抽象的な関数$F(src), F(tgt)$に、それぞれ値の割り当て $e \mapsto i$, $e \mapsto j$ の定義を加えて、具体的な関数にすることができます。

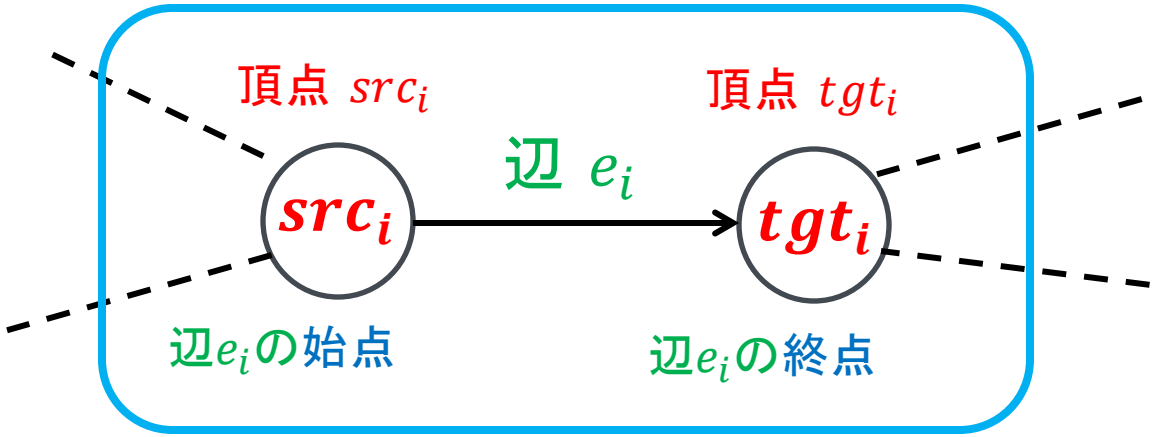
$$F(src) : \{1,2,\dots,m\} \to \{1,2,\dots,n\}$$
$$e \mapsto i$$

$$F(tgt) : \{1,2,\dots,m\} \to \{1,2,\dots,n\}$$
$$e \mapsto j$$

この操作を、グラフの中に含まれる m個の全ての辺について繰り返します。図も、eがm個あることがわかりやすいように、少し書き換えましょう。

一つの辺eについての
部分グラフ





e_iは、m個ある。
i は、1からmまでを走る

関数$F(src)$, $F(tgt)$ の具体的な定義
これがグラフの具体的な形を決める

$$F(src) : F(E) \to F(V)$$
$$e_1 \mapsto src_1$$
$$e_2 \mapsto src_2$$
$$e_3 \mapsto src_3$$
$$\cdots\cdots$$
$$e_m \mapsto src_m$$

$$F(tgt) : F(E) \to F(V)$$
$$e_1 \mapsto tgt_1$$
$$e_2 \mapsto tgt_2$$
$$e_3 \mapsto tgt_3$$
$$\cdots\cdots$$
$$e_m \mapsto tgt_m$$

先の関数$F(src)$, $F(tgt)$ の定義は、
次のような表にまとめることができます。

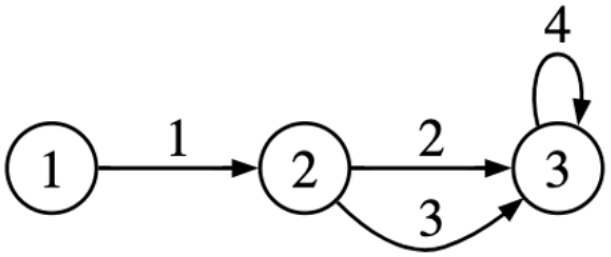
E	src	tgt
e_1	src_1	tgt_1
e_2	src_2	tgt_2
e_3	src_3	tgt_3
…	…	…
e_m	src_m	tgt_m

こうした形で与えられる関数$F(src)$, $F(tgt)$ の定義は、抽象的なスキーマ・カテゴリーCを、具体的なグラフのカテゴリーF(C) = C-setに変換する上で、重要な役割を果たします。

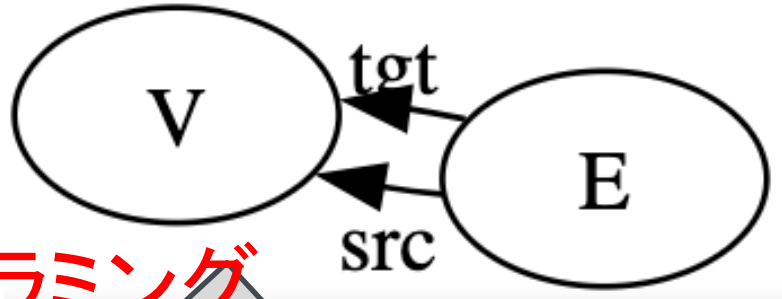
関数$F(src)$, $F(tgt)$ の定義の例

Graph {V:3, E:4}

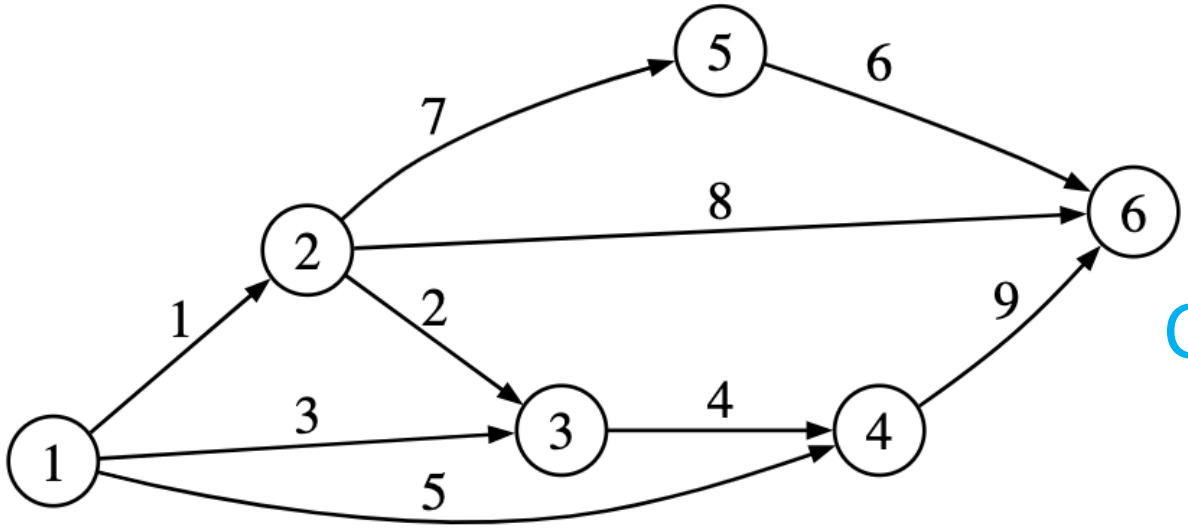
E	src	tgt
1	1	2
2	2	3
3	2	3
4	3	3



CatlabでのC-setプログラミング



Functor



C-set とは何か 2

サンプルで学ぶ

Catlabのプログラム・サンプルから「グラフ＝C-set」を学ぶ

このセッションでは、Catlabではグラフを：どのように扱っているかを、プログラムのサンプルで学びます。

前回のセッションで見たように、Catlab ではグラフの抽象的な定義である「グラフのスキーマ」C に、functor F を適用して具体的なグラフ F(C) を作ろうとします。

F(C) が「具体的」なのは、F(C)のオブジェクトもF(C)の射も、いずれも「集合(Set)」の上で定義されているからです。

F(C)は、どのように「具体的」か

こうした 集合(Set)に値を取る functor F : C → Set (この定義が「C-set」という名前の由来です)のもとでは、F(C)のオブジェクトF(V)は、グラフの頂点を表す自然数の集合に、F(C)のオブジェクトF(E)は、グラフの辺を表す自然数の集合になります。

同様に、F(C)の二つの射 F(src), F(tgt) は、F(E)からF(V)への -- すなわち自然数の集合から自然数の集合への関数になります。

Catlabでの C-setのプログラミングは、グラフの形を決める情報の不足を補う

もっとも、C → Set というfunctorの性質から導かれるこうしたC-set の性質は、まだ、具体的なグラフの形を与えるには十分ではありません。そこには、どの頂点とどの頂点が辺で結ばれているかという、グラフの形を決める情報が不足しています。

Catlabでの「グラフ = C-set」のプログラミングは、こうしたグラフの形を決める情報の不足を補うものです。

具体的には、F(C)の二つの射 F(src), F(tgt) の定義を具体的に与えることで、functor F を「グラフ = C-set」を実現するものにします。

「グラフ ＝ C-set」のプログラミングは、functorのプログラミング

Catlabでのグラフのプログラミングは、抽象的なグラフのスキーマを、具体的な「グラフ ＝ C-set」に変換するfunctor F のプログラミングに他なりません。

今回のセッションでは、いくつかのプログラムの例を紹介するのですが、それらをfunctorのプログラミングだと、是非、意識してみてください。

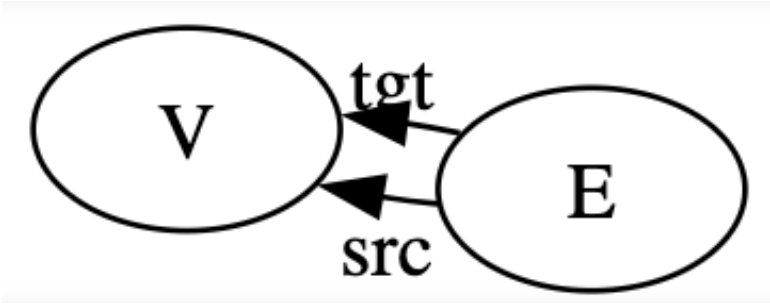
「グラフのスキーマ」の定義を見る

今回のセッションでは、まず最初に、「有向グラフのスキーマ」であるカテゴリーC（ここでは、"SchGraph"という名前を与えられています）の定義を見ていこうと思いますます。

それは、次のようなプログラムになっています。

スキーマCの定義

```
using Catlab.CategoricalAlgebra

@present SchGraph(FreeSchema) begin
  V::Ob
  E::Ob
  src::Hom(E,V)
  tgt::Hom(E,V)
end
```



ここで、V::Ob , E::Ob は、V, E が カテゴリーCのオブジェクトであることを表しています。

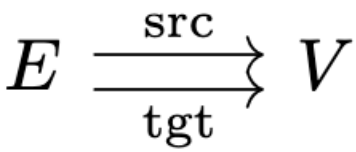
src::Hom(E,V) , tgt::Hom(E,V) は、srcとtgt が、カテゴリーCのEからVに向かう射を表しています。

カテゴリー論の定義、そのままですね。

プログラムの冒頭の "@present"のような'@' で始まる命令は、Julia言語で「マクロ命令」と呼ばれるものなのですが、今回はマクロ命令の説明は行いません。ただ、"@present" マクロは、「スキーマ」の定義に用いられることを、知ってもらえればと思います。

全ての「有向グラフ」を生成するカテゴリー C
スキーマ C

カテゴリー C



全ての「有向グラフ」を生成するカテゴリー C は、次のものからなります。

- Cの二つのオブジェクト: $E, V \in Obj(C)$
  - EはEdgeでグラフの辺の意味
  - VはVertexでグラフの頂点の意味
- Cの二つの射: $src, tgt : E \to V \in Hom(C)$
  src, tgt は、辺が与えられたとき頂点を返す
  - srcはsourceで辺の始点
  - tgtはtargetで辺の終点

プログラム・サンプル

このセッションでは、いくつかのプログラムのサンプルを提供しています。

たくさんサンプルをあげたのですが、実は、まったく同じスタイルのプログラムです。

グラフの形についての基本的な情報

基本になっているのは、前回のセッションの最後にまとめたように、グラフの形を決めるF(src), F(tgt)というC-setの二つの関数の働きは、次のような表にまとめられると言うことです。

E	src	tgt
1	1	2
2	2	3
3	2	3
4	3	3

この表は横に読めば、ある辺の始点と終点が簡単にわかります。

辺1は頂点1から頂点2への矢印であり、辺2は頂点2から頂点3への矢印であり、辺3は頂点2から頂点3への矢印であり、辺4は頂点3から頂点3への矢印である。等々。

一つ注意して欲しいのは、グラフの形を決めるF(src), F(tgt)というC-setの二つの関数の働きを定義するこの表現を、プログラムでは、縦に読む形で表現していると言うことです。

この例では、src = [1, 2, 2, 3] で、tgt = [2, 3, 3, 3] という形で、プログラムに組み込まれています。

二つの頂点と一つの辺を持つ有向グラフ

```
using Catlab.CategoricalAlgebra
using Catlab.Graphs, Catlab.Graphics

g = Graph()
add_parts!(g, :V, 2)
add_parts!(g, :E, 1, src=[1], tgt=[2])
g
```

Graph {V:2, E:1}

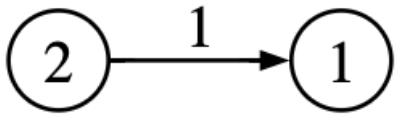
E	src	tgt
1	1	2

```
g = Graph()
add_parts!(g, :V, 2)
add_parts!(g, :E, 1, src=[2], tgt=[1])
g
```

Graph {V:2, E:1}

E	src	tgt
1	2	1

```
to_graphviz(g, node_labels=true, edge_labels=true)
```



二つの頂点と二つの辺を持つ有向グラフ

```
using Catlab.CategoricalAlgebra
using Catlab.Graphs, Catlab.Graphics

g = Graph()
add_parts!(g, :V, 2)
add_parts!(g, :E, 2, src=[1,2], tgt=[2,1])
g
```

Graph {V:2, E:2}

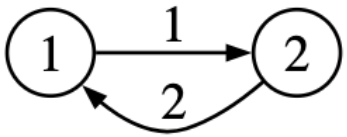
E	src	tgt
1	1	2
2	2	1

```
using Catlab.CategoricalAlgebra
using Catlab.Graphs, Catlab.Graphics

g = Graph()
add_parts!(g, :V, 2)
add_parts!(g, :E, 2, src=[1,2], tgt=[2,1])
g
```

Graph {V:2, E:2}

E	src	tgt
1	1	2
2	2	1

```
to_graphviz(g, node_labels=true, edge_labels=true)
```



二つの頂点と二つの辺を持つ有向グラフ

```
using Catlab.CategoricalAlgebra
using Catlab.Graphs, Catlab.Graphics

g = Graph()
add_parts!(g, :V, 2)
add_parts!(g, :E, 2, src=[1,2], tgt=[2,1])
g
```

Graph {V:2, E:2}

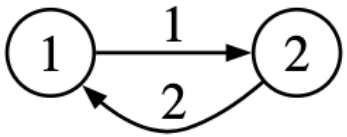
E	src	tgt
1	1	2
2	2	1

```
using Catlab.CategoricalAlgebra
using Catlab.Graphs, Catlab.Graphics

g = Graph()
add_parts!(g, :V, 2)
add_parts!(g, :E, 2, src=[1,2], tgt=[2,1])
g
```

Graph {V:2, E:2}

E	src	tgt
1	1	2
2	2	1

```
to_graphviz(g, node_labels=true, edge_labels=true)
```



二つの頂点と三つの辺を持つ有向グラフ

```
using Catlab.CategoricalAlgebra
using Catlab.Graphs, Catlab.Graphics

g = Graph()
add_parts!(g, :V, 2)
add_parts!(g, :E, 3, src=[1,2,1], tgt=[2,1,1])
g
```

Graph {V:2, E:3}

E	src	tgt
1	1	2
2	2	1
3	1	1

```
using Catlab.CategoricalAlgebra
using Catlab.Graphs, Catlab.Graphics

g = Graph()
add_parts!(g, :V, 2)
add_parts!(g, :E, 3, src=[1,2,1], tgt=[2,1,1])
g
```

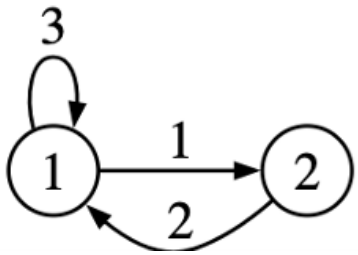
Graph {V:2, E:3}

E	src	tgt
1	1	2
2	2	1
3	1	1

Graph {V:2, E:3}

E	src	tgt
1	1	2
2	2	1
3	1	1

```
to_graphviz(g, node_labels=true, edge_labels=true)
```



二つの頂点と四つの辺を持つ有向グラフ

```
using Catlab.CategoricalAlgebra
using Catlab.Graphs, Catlab.Graphics

g = Graph()
add_parts!(g, :V, 2)
add_parts!(g, :E, 4, src=[1,2,1,1], tgt=[2,1,1,2])
g
```

Graph {V:2, E:4}

E	src	tgt
1	1	2
2	2	1
3	1	1
4	1	2

```
using Catlab.CategoricalAlgebra
using Catlab.Graphs, Catlab.Graphics

g = Graph()
add_parts!(g, :V, 2)
add_parts!(g, :E, 4, src=[1,2,1,1], tgt=[2,1,1,2])
g
```

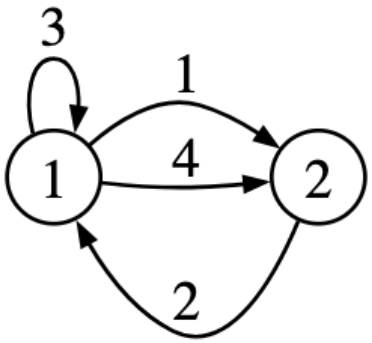
Graph {V:2, E:4}

E	src	tgt
1	1	2
2	2	1
3	1	1
4	1	2

Graph {V:2, E:4}

E	src	tgt
1	1	2
2	2	1
3	1	1
4	1	2

```
to_graphviz(g, node_labels=true, edge_labels=true)
```



三つの頂点と四つの辺を持つ有向グラフ

```
using Catlab.CategoricalAlgebra
using Catlab.Graphs, Catlab.Graphics

g = Graph()
add_parts!(g, :V, 3)
add_parts!(g, :E, 4, src=[1,2,2,3], tgt=[2,3,3,3])
g
```

Graph {V:3, E:4}

E	src	tgt
1	1	2
2	2	3
3	2	3
4	3	3

```
using Catlab.CategoricalAlgebra
using Catlab.Graphs, Catlab.Graphics

g = Graph()
add_parts!(g, :V, 3)
add_parts!(g, :E, 4, src=[1,2,2,3], tgt=[2,3,3,3])
g
```

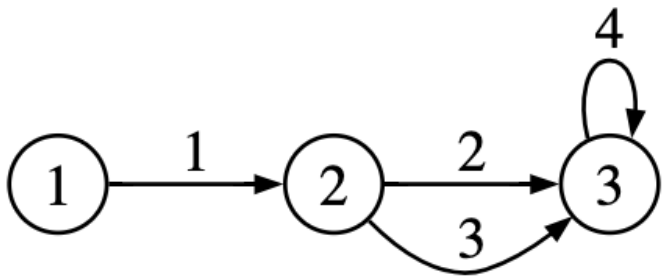
Graph {V:3, E:4}

E	src	tgt
1	1	2
2	2	3
3	2	3
4	3	3

Graph {V:3, E:4}

E	src	tgt
1	1	2
2	2	3
3	2	3
4	3	3

```
to_graphviz(g, node_labels=true, edge_labels=true)
```



```
g = Graph()
add_vertices!(g, 6)
add_edges!(g, [1,2,1,3,1,5,2,2,4], [2,3,3,4,4,6,5,6,6])
g
```

Graph {V:6, E:9}

E	src	tgt
1	1	2
2	2	3
3	1	3
4	3	4
5	1	4
6	5	6
7	2	5
8	2	6
9	4	6

6つの頂点と9つの辺を持つ
有向グラフ

```
g = Graph()
add_vertices!(g, 6)
add_edges!(g, [1,2,1,3,1,5,2,2,4], [2,3,3,4,4,6,5,6,6])
g
```

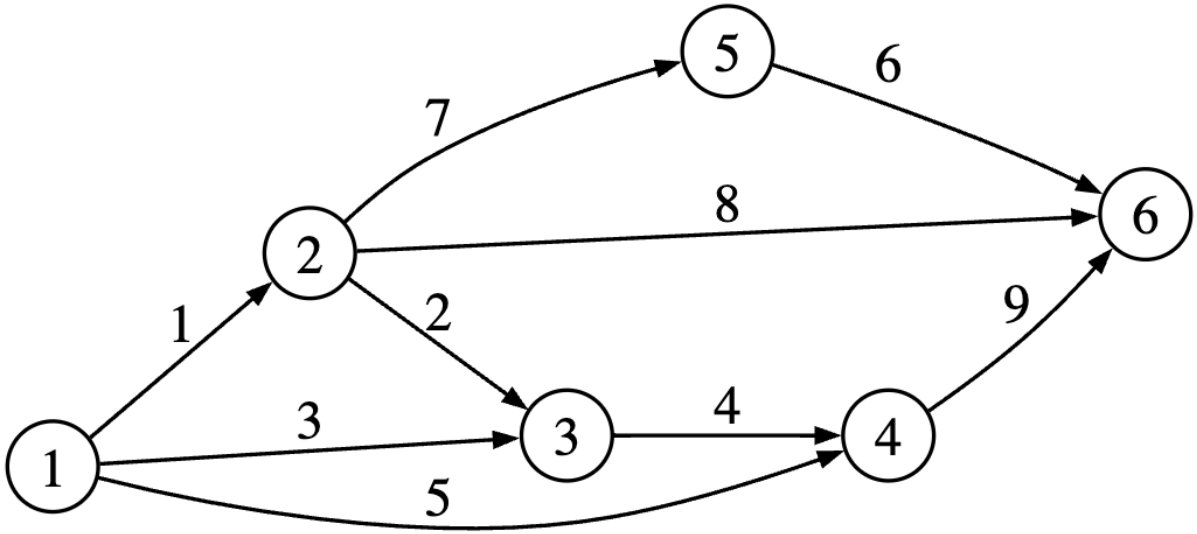
Graph {V:6, E:9}

E	src	tgt
1	1	2
2	2	3
3	1	3
4	3	4
5	1	4
6	5	6
7	2	5
8	2	6
9	4	6

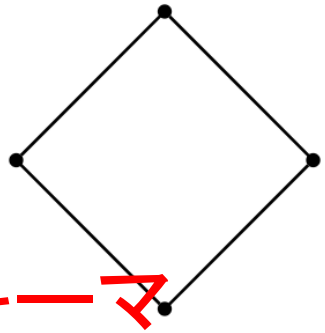
Graph {V:6, E:9}

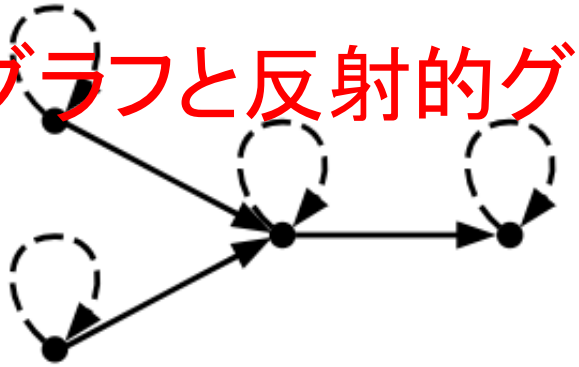
E	src	tgt
1	1	2
2	2	3
3	1	3
4	3	4
5	1	4
6	5	6
7	2	5
8	2	6
9	4	6

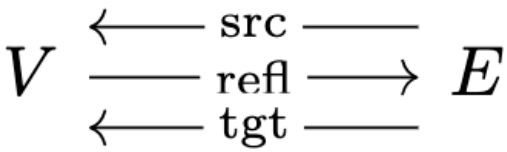
```
to_graphviz(g, node_labels=true, edge_labels=true)
```



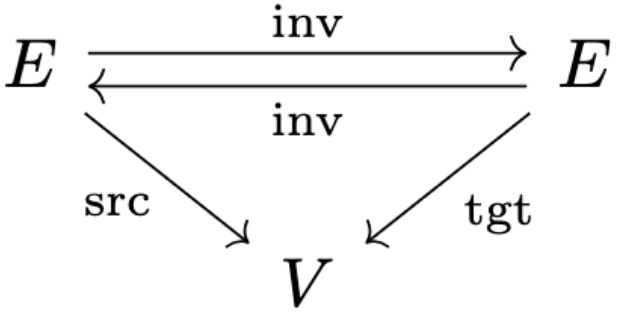
無向グラフと反射的グラフのスキーマ



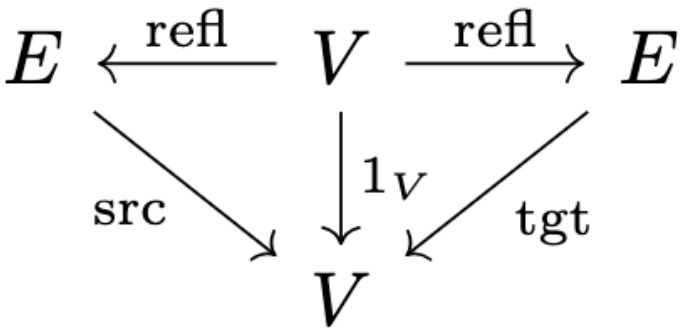




反射的グラフのスキーマ



無向グラフのスキーマ



C-set とは何か 3

スキーマ Cを変える

このセッションの目的

このセッションでは、「グラフ＝C-set」のスキーマ・カテゴリーCを変化させると、生成されるグラフがどのように変わるのかを見てみようと思います。

最初に紹介するのは、「無向グラフ」のスキーマです。

「無向グラフ」は「Symmetric Graph」と呼ばれます。

今まで見てきたスキーム・カテゴリーCでは、「有向グラフ」しか生成できないのです。

次に、全ての頂点が自分自身に帰ってくるループをもつ「Reflexive graphs = 反射的グラフ」のスキーマを考えます。

「反射的グラフ」は「有向グラフ」の一種ですが特別な性質を持っています。

無向グラフ Symmetric Graph

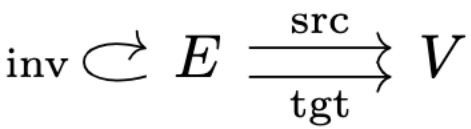
C-setの枠組みは、基本的には、有向グラフを生成するものです。
この枠組みで、どのように、無向グラフを生成するのでしょう？

基本的なアイデアは、全ての辺が反対の向きを持つ二つの辺から構成されているなら、それが無向グラフだと考えることです。

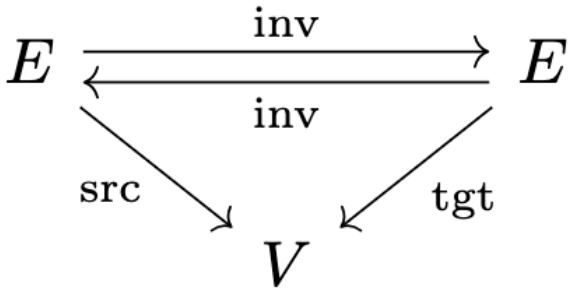
無向グラフのスキーマには、ベースになる有向グラフのスキーマに、全ての辺の上で定義され、その向きを反対にする射inv(involution)が組み込まれています。

$$inv : E \to E$$

無向グラフのスキーマ　SchSymmetricGraph



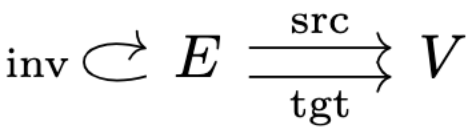
可換図式で書くと



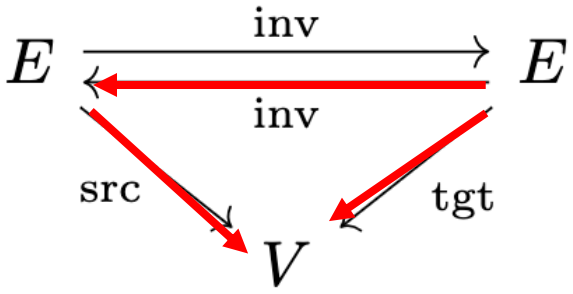
これらの射は、次の条件を満たさねばなりません。

inv·src = tgt, inv·tgt = src, inv·inv = I

無向グラフのスキーマ　SchSymmetricGraph



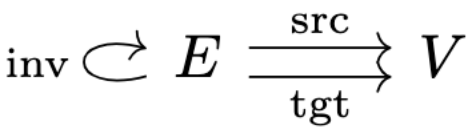
可換図式で書くと



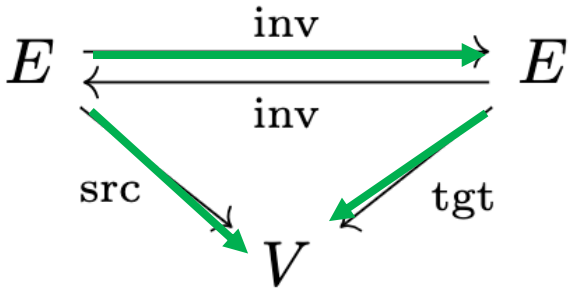
これらの射は、次の条件を満たさねばなりません。

inv·src = tgt, inv·tgt = src, inv·inv = I

無向グラフのスキーマ　SchSymmetricGraph



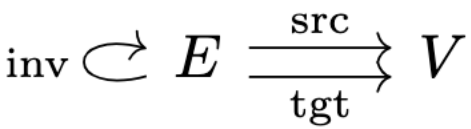
可換図式で書くと



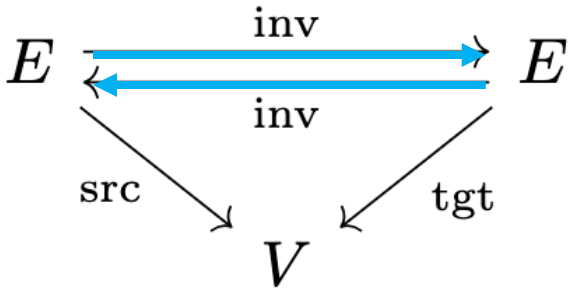
これらの射は、次の条件を満たさねばなりません。

inv·src = tgt, **inv·tgt = src**, inv·inv = I

無向グラフのスキーマ　SchSymmetricGraph



可換図式で書くと



これらの射は、次の条件を満たさねばなりません。

inv·src = tgt, inv·tgt = src, **inv·inv = I**

CatlabでのSchSymmetricGraphの定義

```
@present SchSymmetricGraph <: SchGraph begin
  inv::Hom(E,E)

  compose(inv,src) == tgt
  compose(inv,tgt) == src
  compose(inv,inv) == id(E)
end

@acset_type SymmetricGraph(SchSymmetricGraph, index=[:src])
```

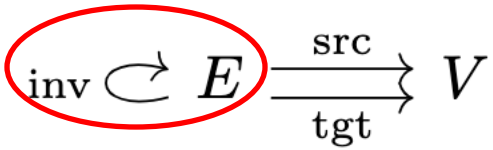
CatlabでのSchSymmetricGraphの定義

```
@present SchSymmetricGraph <: SchGraph begin
  inv::Hom(E,E)

  compose(inv,src) == tgt
  compose(inv,tgt) == src
  compose(inv,inv) == id(E)
end

@acset_type SymmetricGraph(SchSymmetricGraph, index=[:src])
```

```
@present SchGraph(FreeSchema) begin
  V::Ob
  E::Ob
  src::Hom(E,V)
  tgt::Hom(E,V)
end
```

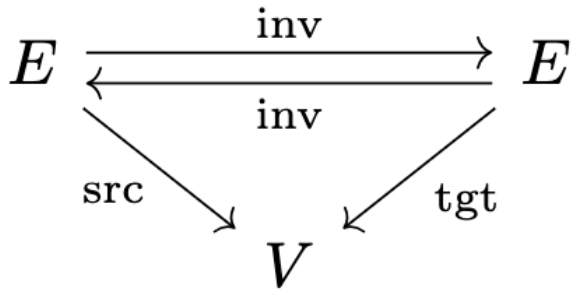
CatlabでのSchSymmetricGraphの定義

```
@present SchSymmetricGraph <: SchGraph begin
  inv::Hom(E,E)      // inv : E → E

  compose(inv,src) == tgt
  compose(inv,tgt) == src
  compose(inv,inv) == id(E)
end

@acset_type SymmetricGraph(SchSymmetricGraph, index=[:src])
```



CatlabでのSchSymmetricGraphの定義

```
@present SchSymmetricGraph <: SchGraph begin
  inv::Hom(E,E)

  compose(inv,src) == tgt
  compose(inv,tgt) == src
  compose(inv,inv) == id(E)
end
```

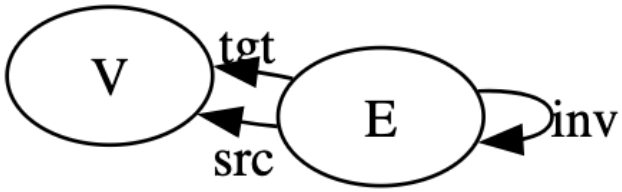



```
@acset_type SymmetricGraph(SchSymmetricGraph, index=[:src])
```

CatlabでのSchSymmetricGraphの定義

```
@present SchSymmetricGraph <: SchGraph begin
  inv::Hom(E,E)

  compose(inv,src) == tgt
  compose(inv,tgt) == src
  compose(inv,inv) == id(E)
end

@acset_type SymmetricGraph(SchSymmetricGraph, index=[:src])
```

to_graphviz(SchSymmetricGraph)



CatlabでのSymmetricGraphのプログラミング

```
using Catlab.Graphs, Catlab.Graphics

g = SymmetricGraph()

add_vertex!(g::AbstractSymmetricGraph; kw...) =
  only(add_vertices!(g, 1; kw...))

add_vertices!(g, 4)
add_edges!(g, [1,2,3,4], [2,3,4,1])
g
```

CatlabでのSymmetricGraphのプログラミング

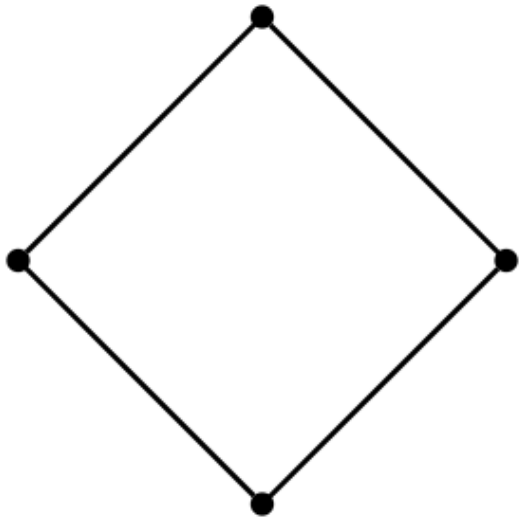
```
using Catlab.Graphs, Catlab.Graphics

g = SymmetricGraph()

add_vertex!(g::AbstractSymmetricGraph; kw...) =
  only(add_vertices!(g, 1; kw...))

add_vertices!(g, 4)
add_edges!(g, [1,2,3,4], [2,3,4,1])
g
```

先のプログラムの出力

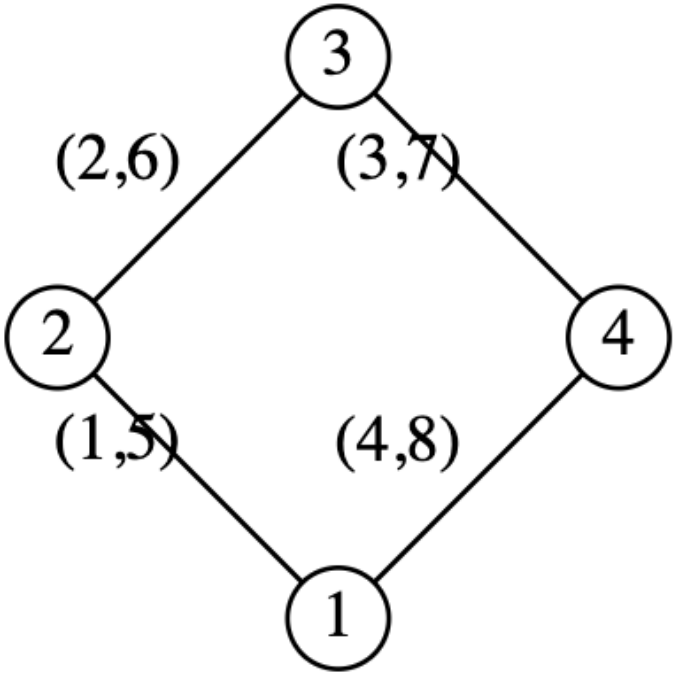
SymmetricGraph {V:4, E:8}

E	src	tgt	inv
1	1	2	5
2	2	3	6
3	3	4	7
4	4	1	8
5	2	1	1
6	3	2	2
7	4	3	3
8	1	4	4

```
to_graphviz(g, prog="circo")
```



```
to_graphviz(g, prog="circo", node_labels=true,
edge_labels=true)
```



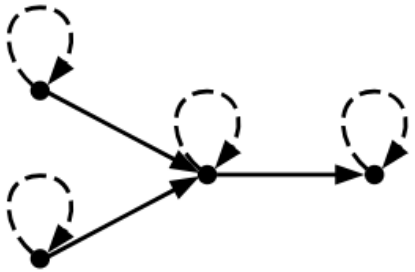
SymmetricGraph {V:4, E:8}

E	src	tgt	inv
1	1	2	5
2	2	3	6
3	3	4	7
4	4	1	8
5	2	1	1
6	3	2	2
7	4	3	3
8	1	4	4

このように、無向グラフは、SchSymmetricGraphをスキーマとした SchSymmetricGraph-set として表現することができます。

反射的グラフ Reflexive graphs

反射的グラフとは、すべての頂点が次のように自己ループを持つグラフのことです。



一見すると、自己ループは取るに足らない追加に見えるかもしれません。反射的グラフは、通常の有向グラフと多かれ少なかれ互換性があります。

しかし、反射的グラフの射はグラフの射とは大きく異なり、その結果、反射的グラフのカテゴリーは有向グラフのカテゴリーとは異なる振る舞いをします。

反射的グラフの「幾何学的」性質

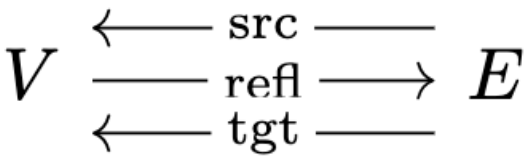
反射的グラフは、さまざまな点でグラフよりも「幾何学的」であることがわかります。

例えば、グラフGとHが連続空間XとYを離散化する場合、グラフの積 G×Hは、グラフが反射的グラフである場合にのみ、積空間X×Yを離散化します。

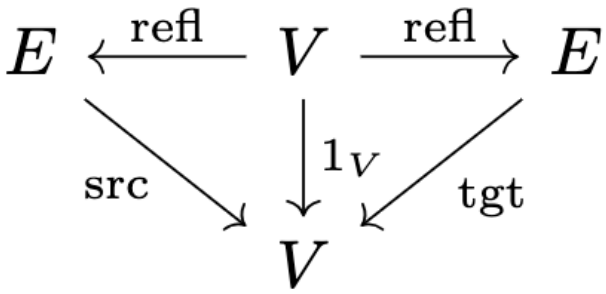
グラフの積とグラフの準同型については、次回のセッションで扱います。

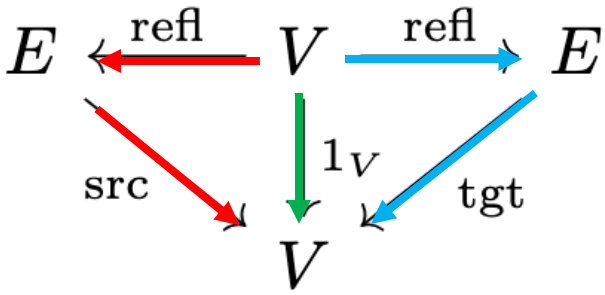
Reflexive graphsのスキーマ

反射的グラフとは、グラフGと各頂点vにvにおける自己ループを割り当てる関数 G(refl) : G(V) → G(E)を持つSch(RGraph)-setです。ここでSch(RGraph)は再帰グラフのスキーマで、次のような形をもつものとして定義されます。



可換図式で示せば、次のような形になります。





refl·src = 1_v refl·tgt = 1_v

Catlabでの反射的グラフのスキーマの定義

```
@present SchReflexiveGraph <: SchGraph begin
  refl::Hom(V,E)

  compose(refl, src) == id(V)
  compose(refl, tgt) == id(V)
end
```

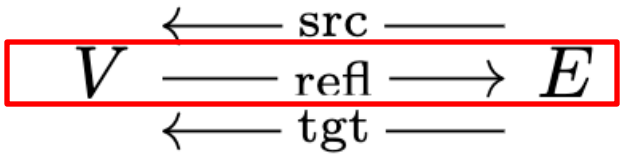
Catlabでの反射的グラフのスキーマの定義

```
@present SchReflexiveGraph <: SchGraph begin
  refl::Hom(V,E)

  compose(refl, src) == id(V)
  compose(refl, tgt) == id(V)
end
```

```
@present SchGraph(FreeSchema) begin
  V::Ob
  E::Ob
  src::Hom(E,V)
  tgt::Hom(E,V)
end
```

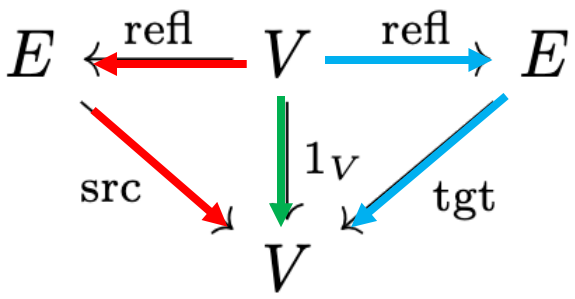
Catlabでの反射的グラフのスキーマの定義

```
@present SchReflexiveGraph <: SchGraph begin
  refl::Hom(V,E)

  compose(refl, src) == id(V)
  compose(refl, tgt) == id(V)
end
```



Catlabでの反射的グラフのスキーマの定義

```
@present SchReflexiveGraph <: SchGraph begin
  refl::Hom(V,E)

  compose(refl, src) == id(V)
  compose(refl, tgt) == id(V)
end
```



グラフC-setからグラフのカテゴリーC-Setへ

このように、反射的グラフは、SchReflexiveGraphをスキーマとした SchReflexiveGraph-set として表現することができます。

スキーマCとfunctor F によるグラフの生成という「グラフ＝C-set」という枠組みは、スキーマCを変化させることによって、多様なグラフの特徴づけに利用することができます。

このアプローチは、C-setであるグラフたちをオブジェクトとし、それらを生成するfunctor間のnatural transformation を射とするFunctorカテゴリー C-Set へと拡大していきます。


Part 4
物理シミレーションとグラフ

Agenda Part 4
物理シミレーションとグラフ

- AlgebraicJuliaとは何か？
- 「AIと物理」というテーマの重要性
- Decapodes プロジェクトとは何か
- 単純な調和振動子の式をグラフで表す
- “A diagrammatic view of differential equations in physics”

AlgebraicJuliaとは何か？

AlgebraicJuliaは、「科学の計算に応用カテゴリー論に基づいた新しいアプローチを産み出すことを狙いとする」というミッションを掲げた Julia言語で書かれたパッケージ群です。

それはまた「科学技術計算に構成可能性(Compositionality)を持ち込む」ことを目標に掲げています。「応用カテゴリー論に基づく」というのは、そういうことです。

AlgebraicJuliaのミッション

Mission

AlgebraicJulia aims to create novel approaches to scientific computing based on applied category theory

科学の計算に応用カテゴリー論に基づいた新しいアプローチを産み出すことを目標とする

https://www.algebraicjulia.org/

AlgebraicJuliaのパッケージ群とCatlab

https://github.com/AlgebraicJulia/Catlab.jl

AlgebraicJuliaは多くのパッケージから構成されています。

その中核をなすのは、科学・技術分野へのカテゴリー論の応用のためのフレームワーク -- プログラミング・ライブラリーとインタラクティブなインターフェースを提供しているCatlabと見做してもいいと思います。

CatlabはNetworkXやGraphvizのように、グラフを描くことを主目的としてつくられたものではありません。(Catlabはrendererとしてgraphvizを*使います。)

カテゴリー論とのインタラクティブなインターフェースを提供するためには、グラフを扱う能力が必要だったからです。

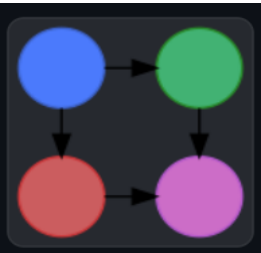
Gatlab

https://github.com/AlgebraicJulia/GATlab.jl

AlgebraicJuliaには、カテゴリー論的に重要な理論展開をしているパッケージが含まれています。それがGatlab (Generalized Algebraic Theories)です。

GATは、コンピュータ・サイエンスの型理論と数学の代数と連携させたものです。それは、コンピュータ上で代数システムを構築する絶好の場所になります。Gatlabは、Catlabの実装に理論的な基礎を提供しています。

"GATlab: Modeling and Programming with Generalized Algebraic Theories" https://arxiv.org/abs/2404.04837



AlgebraicJulia

An Ecosystem of Software Based on Generalized Algebra and Category Theory in Julia

147 followers https://www.algebraicjulia.org

Overview Repositories 52 Projects Packages People 8

Pinned

Catlab

Catlab.jl Public

A framework for applied category theory in the Julia language

Julia 601 56

ACSets.jl Public

ACSets: Algebraic databases as in-memory data structures

Julia 17 7

Gatlab

Decapodes.jl Public

A framework for composing and simulating multiphysics systems

Julia 46 14

GATlab.jl Public

GATlab: a computer algebra system based on generalized algebraic theories (GATs)

Julia 21 2

AlgebraicPetri.jl Public

Build Petri net models compositionally

Julia 74 20

AlgebraicRewriting.jl Public

Implementations of algebraic rewriting techniques like DPO, SPO, SqPO.

https://github.com/AlgebraicJulia

連続セミナーとAlgebraicJulia

連続セミナーのコンテンツは、AlgebraicJuliaのパッケージの展開と結びついています。

Part 3で取り上げたの、次のものです。

- グラフのC-Set(あるいは、AC-Set)としての特徴づけ
  ACSets
  https://github.com/AlgebraicJulia/ACSets.jl

次回以降、このパッケージの紹介をしようと思います。

- Agent-Base Modelのコンピュータ上での実装
  AlgebraicABMs
  https://github.com/AlgebraicJulia/AlgebraicABMs.jl

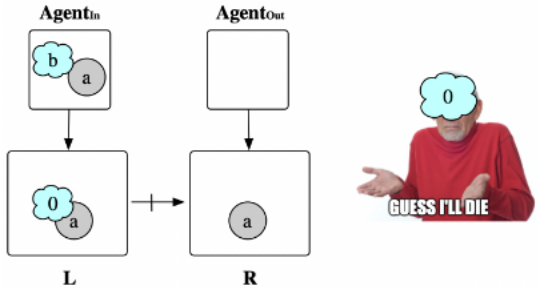
セミナーのトピックスとAlgebraicJulia.blog

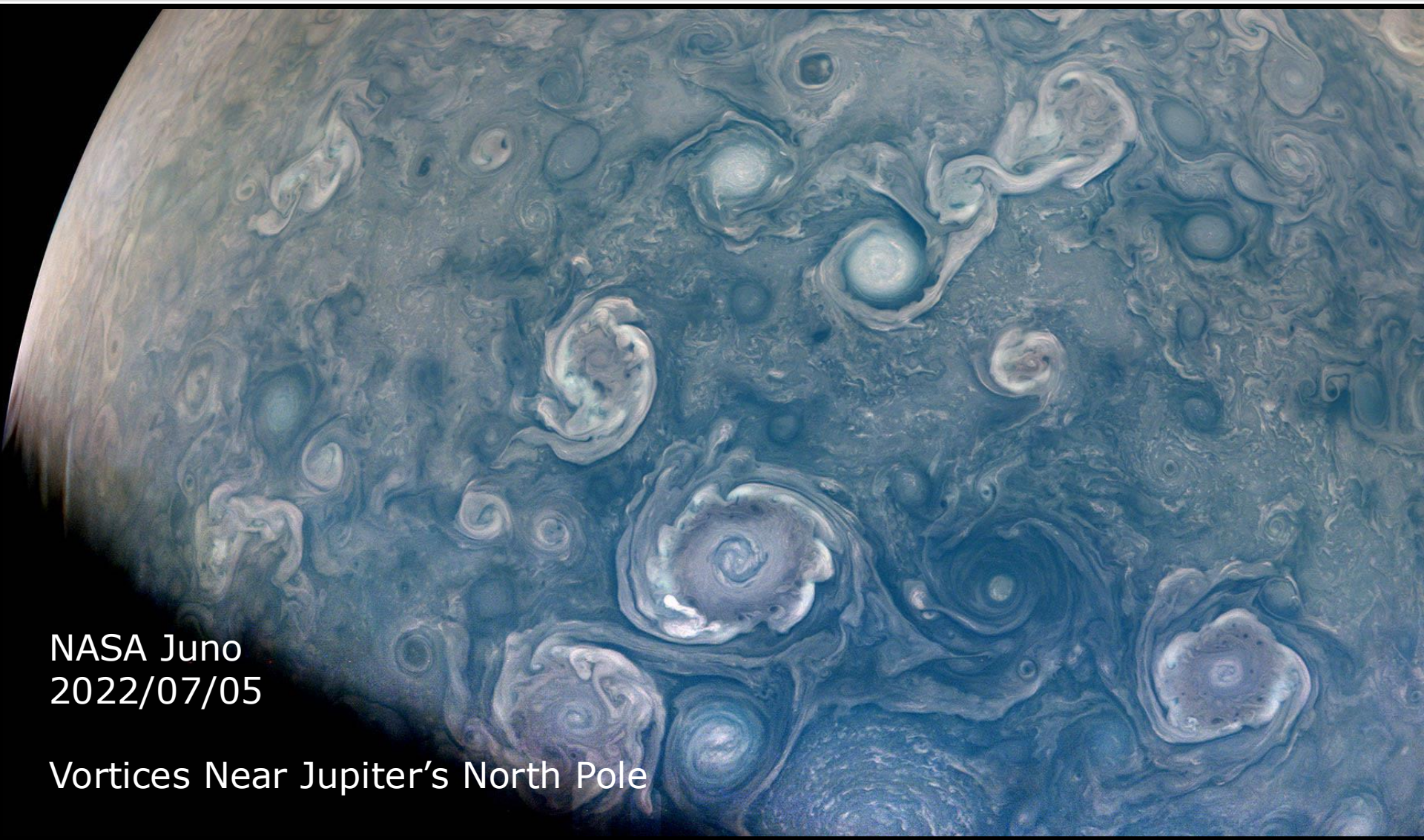
GiTHubやarXiv以外に、AlgebraicJulia.blog https://blog.algebraicjulia.org/ に、有益な情報がたくさんあります。セミナーとの関連では、つぎのblogを参照ください。

- グラフとCsetのトピックスについては、Evan Pattersonの https://blog.algebraicjulia.org/#category=graphs
- Agent-Based Model については、Kris Brownの https://blog.algebraicjulia.org/post/2023/07/graphical-schedule/

Agent-based modeling via graph rewriting

REWRITING | ATTRIBUTED-C-SETS | MODELS | DYNAMICAL SYSTEMS





NASA Juno
2022/07/05

Vortices Near Jupiter’s North Pole

物理シミレーションとグラフ

「AIと物理」というテーマの重要性

このセッションでは、「コンピュータ・シミレーションとグラフ」の話をしようと思います。

多分、僕は、問題を機械の「言語能力と数学的能力」に還元する傾向があるのですが、それはある意味では狭い見方かもしれません。現在の「生成AI」の「限界」を現実的・具体的に考えるのなら、「AIと数学」だけでなく、「AIと科学技術」というテーマ、その中でも特に、「AIと物理学」というテーマは重要です。

皆さんの中には、物理学に興味をお持ちの方も多いと思います。ぜひ、今回のセッションの内容から「AIと数学と物理学」について考えてもらえたら嬉しいです。

今回のPart 4でとりあげること
木星の北極には巨大な渦があるということ

木星の渦というと赤道ちかくの「大赤斑」が有名ですが、NASAの木星探査船 Juno は、何度かのSwing By を繰り返して木星に接近し、木星の北極近くに巨大な渦が存在することを発見しました。それればかりでなく、その渦が同一緯度上に規則的に配置されていること見出しました。

その理由については、いろいろな説があったのですが、近年、それを明確に説明する理論モデルが提出されました。ただ、今回のInterludeで紹介するのは、その理論モデルについてではありません（僕は、よく理解していないし。）

今回紹介するのは、そうした渦の振る舞いを再現するコンピュータ・シミレーションができあがったということです。

それを可能にしたのは、AlgeraicJuliaのパッケージの一つである、Decapodes.jl でした。

Decapodes

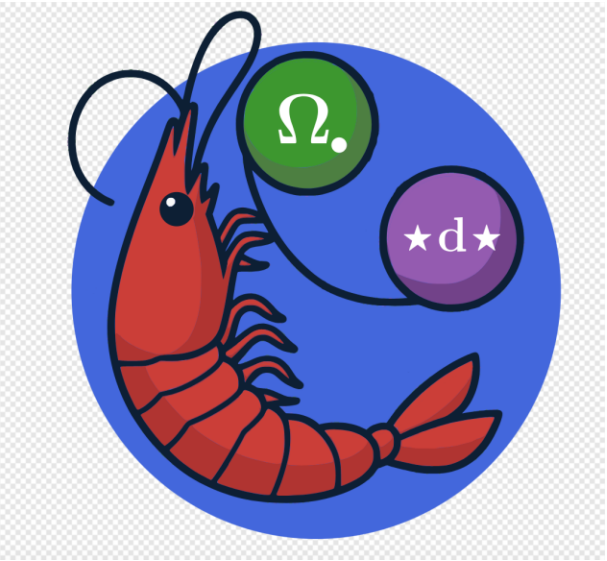
＝

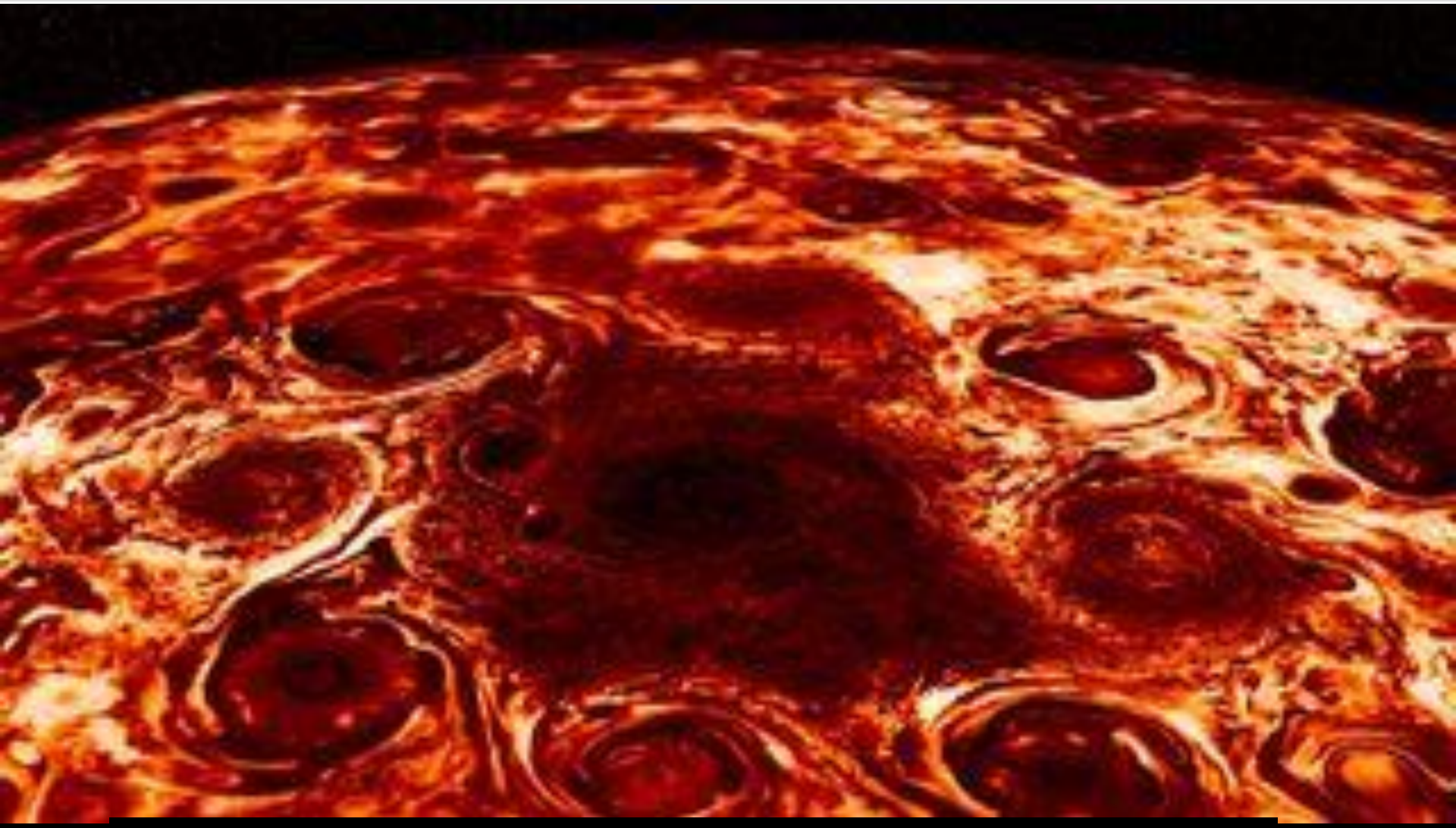
十脚

＝

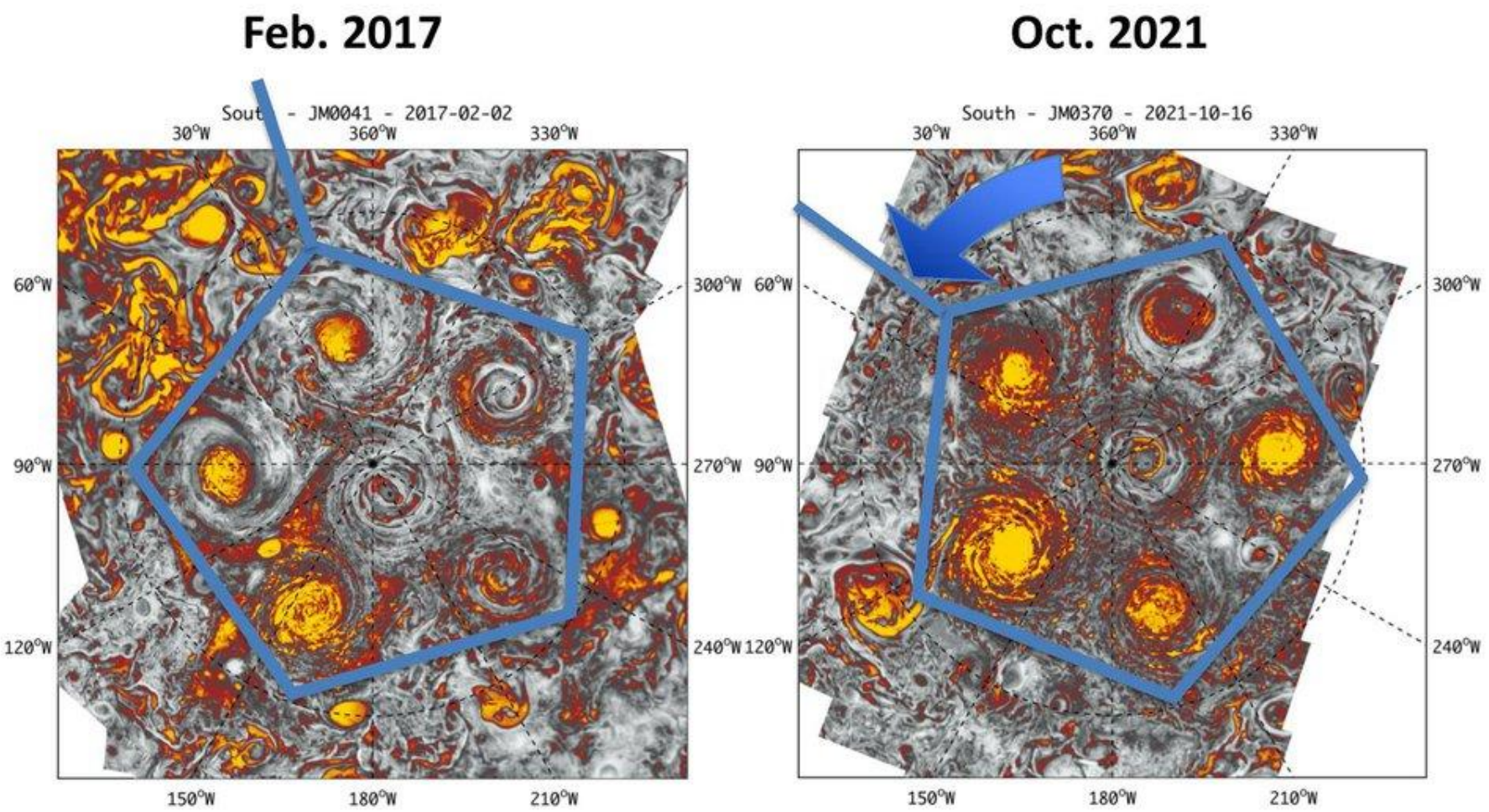
さくらえび

甲殻類プロジェクト？





This composite image from the Jovian Infrared Auroral Mapper (JIRAM) instrument on NASA's Juno spacecraft shows the central cyclone at Jupiter's north pole and the eight cyclones that encircle it.
(Image credit: NASA/JPL-Caltech/SwRI/ASI/INAF/JIRAM
https://www.space.com/jupiter-polar-vortices-stability.html



This annotated composite image depicts the movement of the polar and circumpolar cyclones of Jupiter's south pole between 2016 (left) and 2021 (right) as seen by the Jovian Infrared Auroral Mapper (JIRAM) instrument aboard NASA's Juno spacecraft. In both images, five cyclones are arranged as a pentagon, with a sixth cyclone in the center (south pole). https://www.jpl.nasa.gov/images/pia24967-jupiters-polar-vortices-over-five-years

nature astronomy

Article | Published: 22 September 2022

Vorticity and divergence at scales down to 200 km within and around the polar cyclones of Jupiter

Andrew P. Ingersoll ✉, Shawn P. Ewald, Federico Tosi, Alberto Adriani, Alessandro Mura, Davide Grassi, Christina Plainaki, Giuseppe Sindoni, Cheng Li, Lia Siegelman, Patrice Klein & William R. Young

Nature Astronomy **6**, 1280–1286 (2022) | Cite this article

https://www.nature.com/articles/s41550-022-01774-0

abstract

2022/09/22

2017年以来、探査機ジュノーは木星の北極で、多角形に配置された8つの小さなサイクロンに囲まれたサイクロンを観測している。なぜこのような配置が安定しているのか、どのように維持されているのかは明らかになっていない。

ここでは、ジュノーに搭載されたJIRAMマッピングスペクトロメーターによって得られた時系列の画像を用いて、風を追跡し、極低気圧とその周辺の2つの低気圧の渦度と水平発散度を測定した。極低気圧とその周辺の低気圧の間に高気圧性リングを発見し、多角形パターンの安定性にはこのような遮蔽が必要であるという説を支持した。

しかし、最小の空間スケール(180km)でも、対流の予想される発散と反サイクロン渦度の空間的相関-は見られなかった。

我々は、木星の対流嵐の大きさが地球と比較して相対的に小さいことが、2つの研究を調和させる可能性があることを示唆する。

American Meteorological Society

2024/06/27

15.2 - The Motion of Jupiter's Polar Cyclones Explained By Vorticity Dynamics and a Center-of-Mass Approach

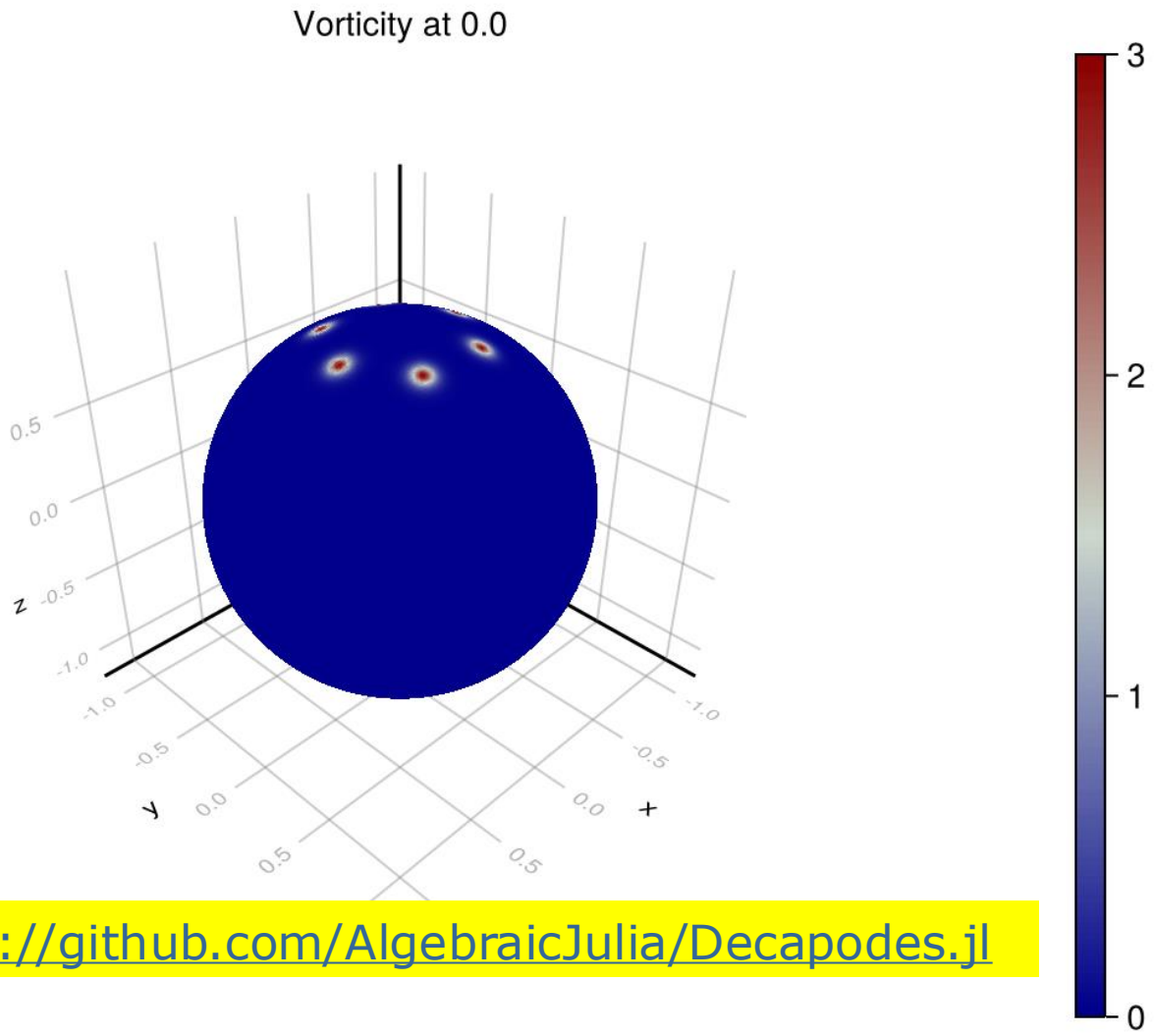
Thursday, June 27, 2024

1:45 PM - 2:00 PM

Adirondack ABC (Hilton Burlington Lake Champlain)

https://ams.confex.com/ams/24Fluid22Middle/meetingapp.cgi/Paper/443600

AlgebraicJulia
Decapodes プロジェクトでのシミレーション



https://github.com/AlgebraicJulia/Decapodes.jl

AlgebraicJulia
Decapodes プロジェクトとは何か

Decapodes.jlは、物理システムを開発、構成、シミュレーションするためのフレームワークです。

Decapodes.jlは、物理方程式を形式化し構成するための応用カテゴリー理論（ ACT = Applied Category Theory）のテクニックと、微分演算子を形式化するための離散外積代数（ DEC = Descrete Exterior Calculus）のテクニックを統合したものです。

CombinatorialSpaces.jlは、空間を離散化し、DEC演算子を単体複体上で定義するためのツールをホストしており、DiagrammaticEquations.jlは、方程式を形式的なACTダイアグラムとして表現するためのツールをホストしています。

このリポジトリはこれら2つのパッケージを組み合わせ、ダイアグラムをシミュレーション可能なコードにコンパイルします。

ACTとDECを組み合わせることで、科学計算のワークフローを改善します。Decapodesのシミュレーションは、階層的に構成可能で、あらゆる種類の多様体に対して一般化でき、人間が読める宣言的なドメイン固有言語（DSL）を用いて、性能と精度を向上させます。

単純な調和振動子の式をグラフで表す

```
using Catlab
using Catlab.Graphics
using DiagrammaticEquations
using DiagrammaticEquations.Deca

oscillator = @decapode begin
  X::Form0
  V::Form0

  ∂ₜ(X) == V
  ∂ₜ(V) == -k(X)
end
```

デフォルトでは、ACSet として表形式で出力される。
表は、変数(X)とその型(Form0)を格納するためのVar。

Var	type	name
1	Form0	X
2	Form0	V
3	infer	•1
4	infer	V

TVarはダイナミクスの接変数(tangent variable) である変数のサブセットを識別するためのものである。

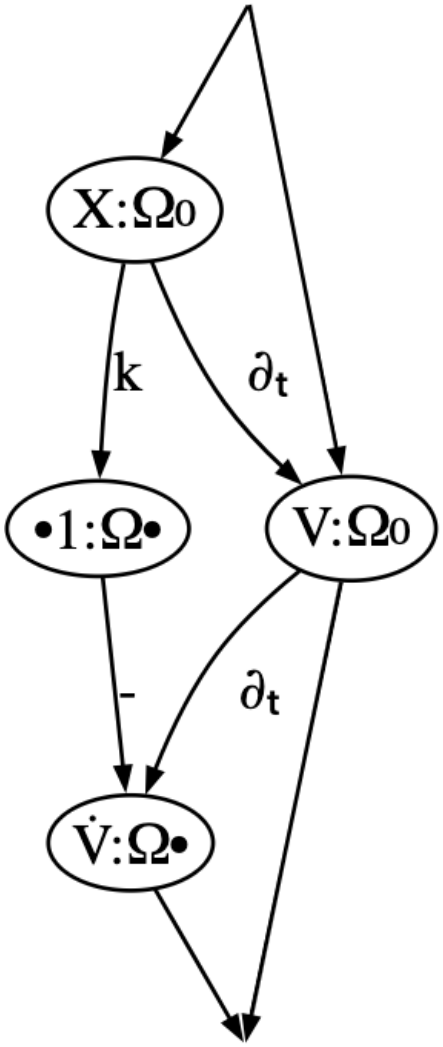
TVar	incl
1	2
2	4

単項演算子は Op1 に、二項演算子は Op2 に格納される。表が空の場合は印刷されない。

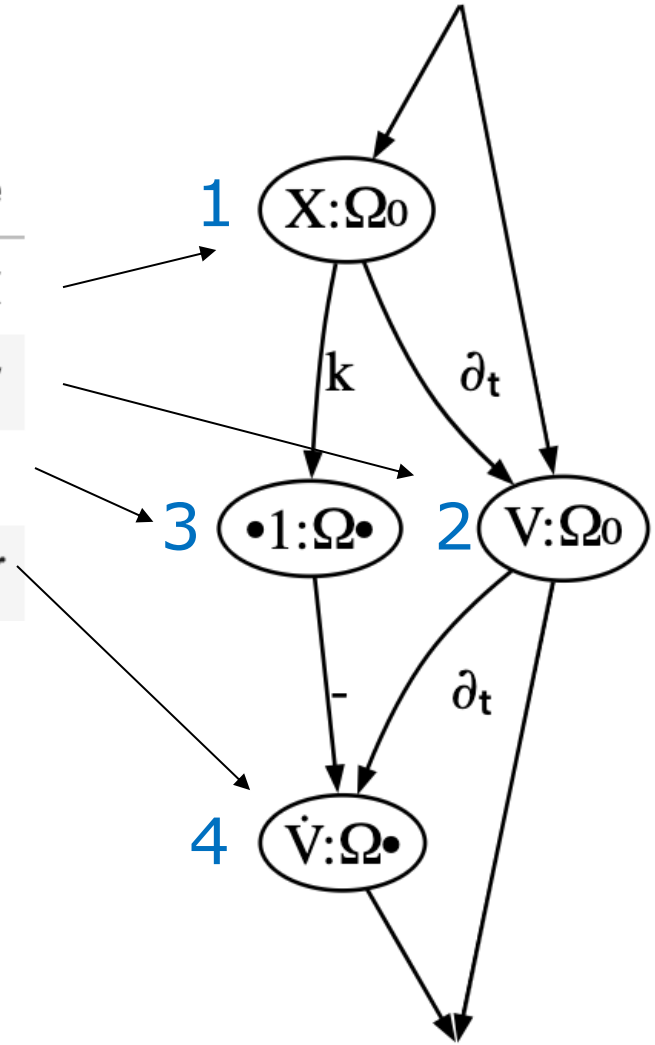
Op1	src	tgt	op1
1	1	2	∂_t
2	2	4	∂_t
3	1	3	k
4	3	4	-

連立方程式はグラフのようなものだが、アリティ（入力の数）とコアリティ（出力の数）は演算子の定義に組み込まれているため、エッジテーブルはない。

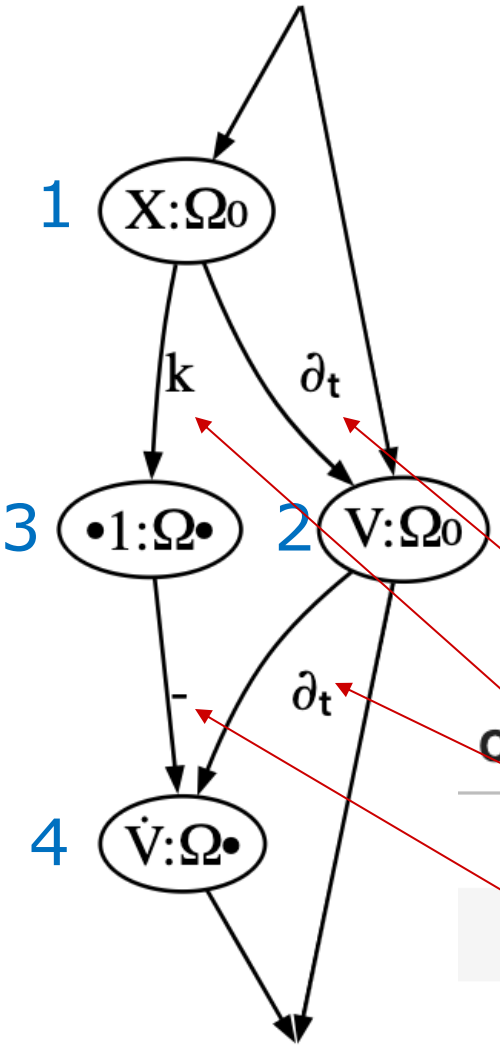
また、出力を有向グラフとして見ることもできる。入力の矢印はシステムの状態変数を指し、出力変数は接変数から指している。dX/dtは$\dot{x}$、d$\dot{x}$は$\ddot{x}$と呼ぶべきことがわかる


X:Ω0
k
∂t
•1:Ω•
V:Ω0
-
∂t
V̇:Ω•

Var	type	name
1	Form0	X
2	Form0	V
3	infer	•1
4	infer	V̇



Var	type	name
1	Form0	X
2	Form0	V
3	infer	•1
4	infer	V̇



Op1	src	tgt	op1
1	1	2	∂t
2	2	4	∂t
3	1	3	k
4	3	4	-

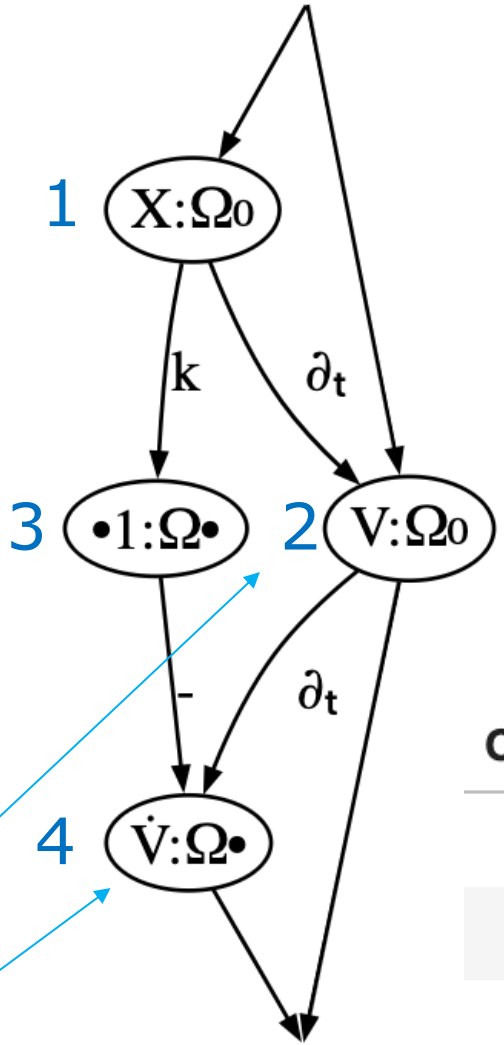
$\partial_t(X) == V$

$\partial_t(V) ==(4)$

$k*X ==(3)$

$-1*(3) ==(4)$

Var	type	name
1	Form0	X
2	Form0	V
3	infer	•1
4	infer	V̇



TVar	incl
1	2
2	4

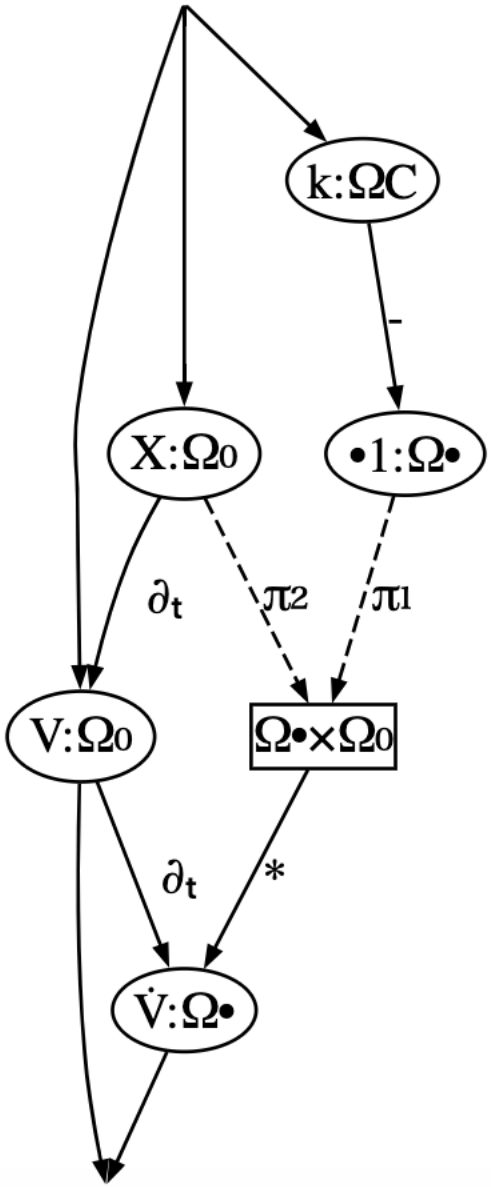
Op1	src	tgt	op1	
1	1	2	∂_t	$\partial_t(X) == V$
2	2	4	∂_t	$\partial_t(V) ==(4)$
3	1	3	k	k*X ==(3)
4	3	4	-	-1*(3) ==(4)

kを定数とした定義

```
oscillator = @decapode begin
  X::Form0
  V::Form0

  k::Constant

  ∂ₜ(X) == V
  ∂ₜ(V) == -k*(X)
end
```

kを定数とした定義のグラフ

```
oscillator = @decapode begin
  X::Form0
  V::Form0

  k::Constant

  ∂ₜ(X) == V
  ∂ₜ(V) == -k*(X)
end
```



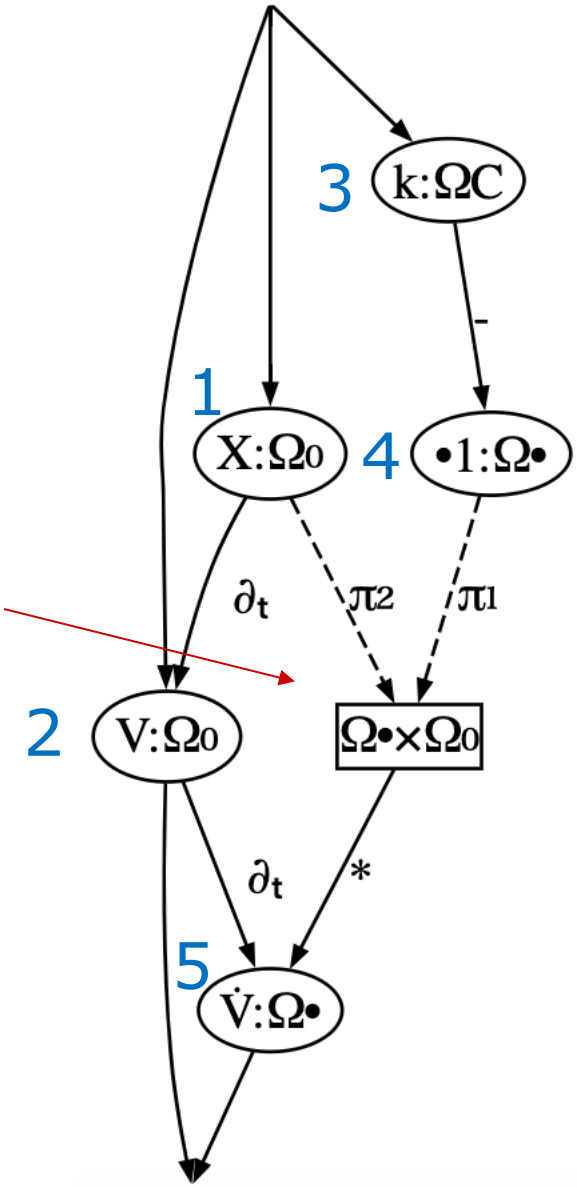
kを定数とした定義のグラフ

Var	type	name
1	Form0	X
2	Form0	V
3	Constant	k
4	infer	•1
5	infer	V̇

Op1	src	tgt	op1
1	1	2	∂_t
2	2	5	∂_t
3	3	4	-

Op2	proj1	proj2	res	op2
1	4	1	5	*

TVar	incl
1	2
2	5

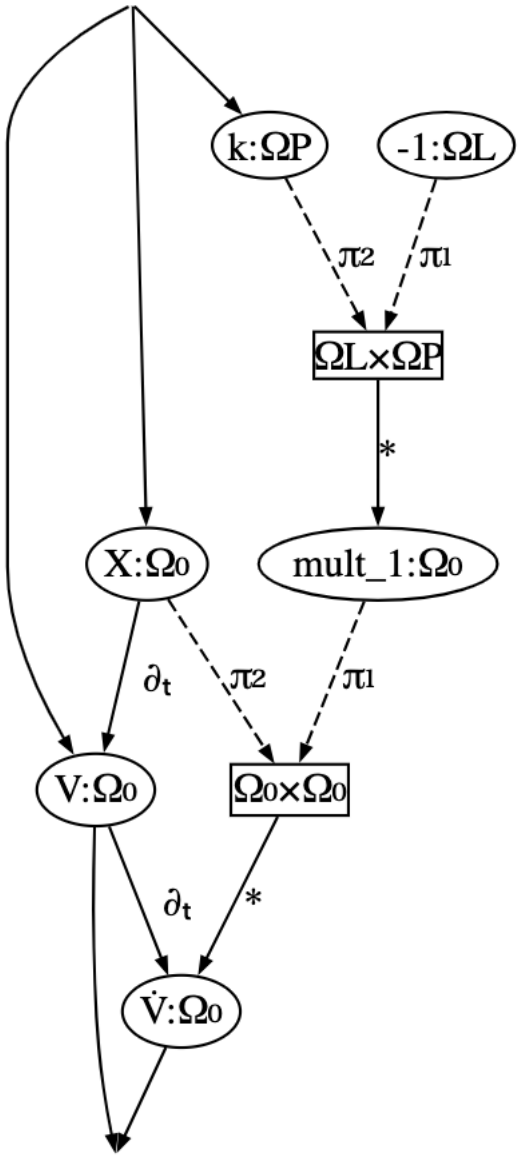


kをパラメータとした定義

```
oscillator = @decapode begin
  X::Form0
  V::Form0

  k::Parameter

  ∂ₜ(X) == V
  ∂ₜ(V) == -k*(X)
end
```

kをパラメータとした定義のグラフ

```
oscillator = @decapode begin
  X::Form0
  V::Form0

  k::Parameter

  ∂ₜ(X) == V
  ∂ₜ(V) == -k*(X)
end
```



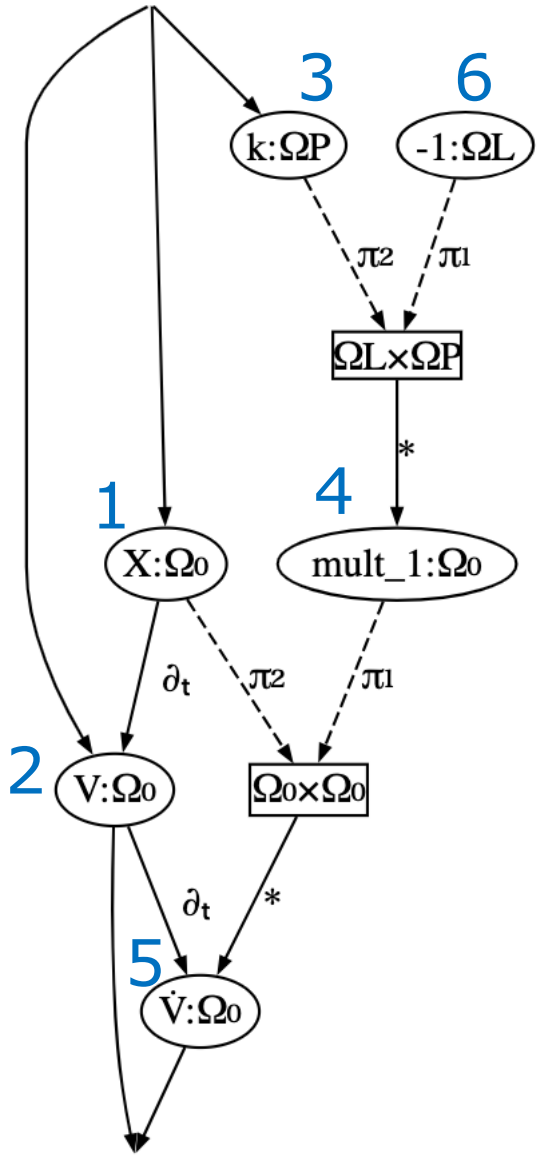
kをパラメータとした定義のグラフ

Var	type	name
1	Form0	X
2	Form0	V
3	Parameter	k
4	infer	mult_1
5	infer	V̇
6	Literal	-1

TVar	incl
1	2
2	5

Op1	src	tgt	op1
1	1	2	∂_t
2	2	5	∂_t

Op2	proj1	proj2	res	op2
1	6	3	4	*
2	4	1	5	*



“A diagrammatic view of differential equations in physics”

微分方程式系をダイアグラムの形で表現することは、物理学のある分野、特に電磁気学や計算物理学では一般的になっている。

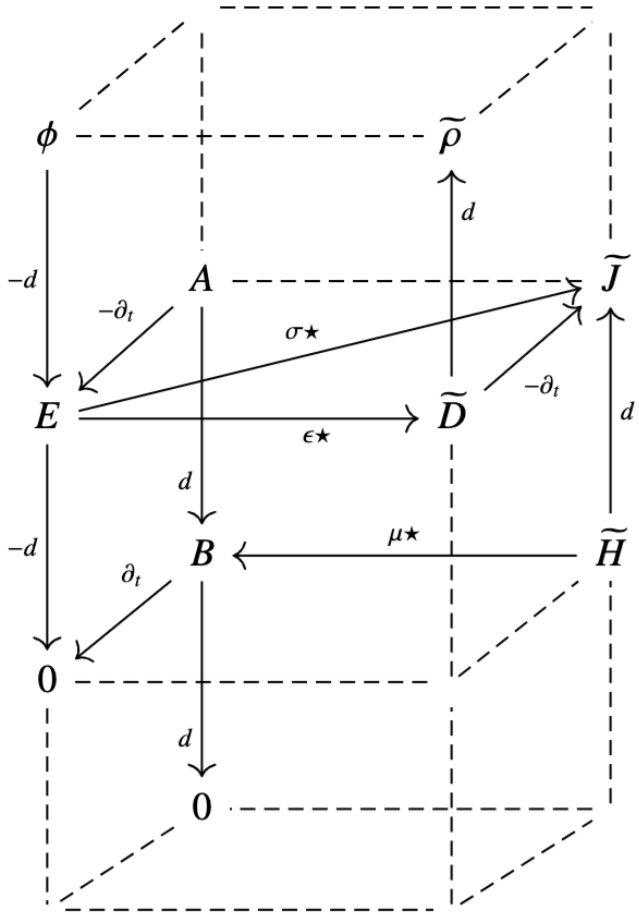
本研究では、このようなダイアグラムの利用を確固たる数学的基盤の上に置くと同時に、方程式系とその解について形式的に推論するための、広く適用可能な枠組みを体系化することを目的とする。

我々の主な数学的手段は、よく知られているカテゴリー論的なダイアグラムと、あまり評価されてこなかったダイアグラム間の射である。

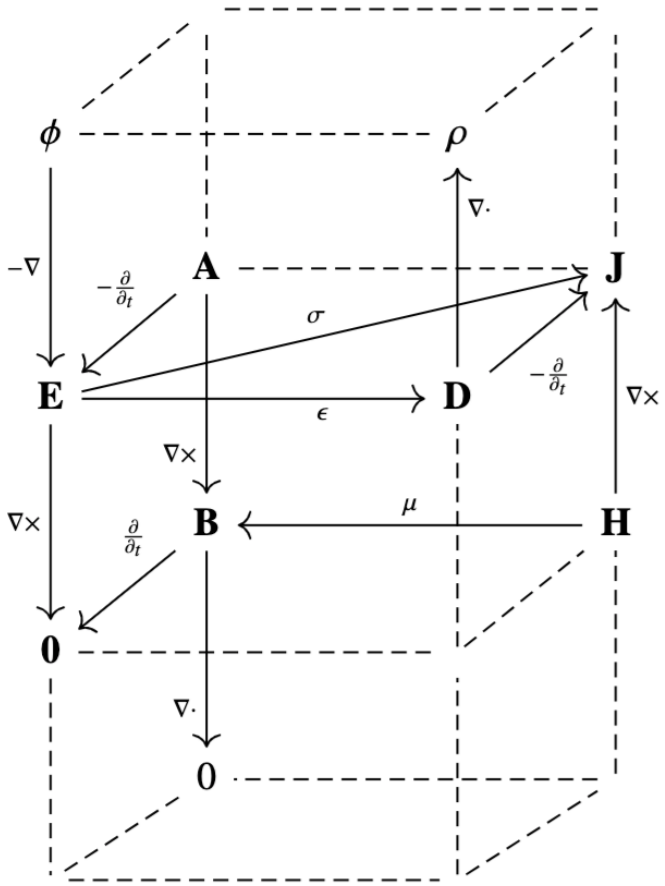
ダイアグラムの枠組みの応用として、複雑な多物理系を基本的な物理原理からどのようにモジュール化できるかを示す。電磁気学、輸送現象、流体力学、その他の分野から引き出された豊富な例が含まれている。

https://arxiv.org/abs/2204.01843

Tonti図 (1972),
the mathematical structure of a large class of physical theories



(a) Maxwell's house in exterior calculus



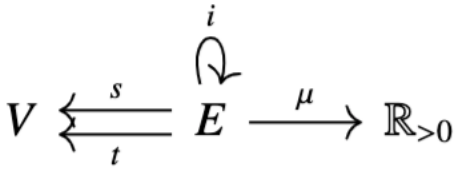
(b) Maxwell's house in vector calculus

Maxwell's equations. An arrow $x \xrightarrow{f} y$ asserts the equation $y = f(x)$. For instance, the arrow $A \xrightarrow{d} B$ asserts that the magnetic field B, a 2-form, and the magnetic potential A, a 1-form, are related by the equation $B = dA$, where d is the exterior derivative. Moreover, a node with multiple incoming arrows

Definition 2.1 (Diagram)**.** A *diagram* in a category C is a functor $D : \mathsf{J} \to \mathsf{C}$, where J, the *shape* or *indexing category* of the diagram, is a small category.

Example 2.2 (Discrete heat equation). The classical heat equation has a discrete analogue on graphs.

Let G be a symmetric weighted graph: a diagram of sets and functions



With this setup, the *discrete heat equation* on the graph G is

$$\partial_n u = \Delta u, \qquad u \in \mathbb{R}^{\mathbb{N}\times V}. \tag{2.1}$$

When $G = \mathbb{Z}^d$, this equation is a discrete version of the heat equation in $\mathbb{R}^d$ [48, §1.3]. We can present the discrete heat equation via the diagram

$$\mathbb{R}^{\mathbb{N}\times V} \underset{\Delta}{\overset{\partial_n}{\rightrightarrows}} \mathbb{R}^{\mathbb{N}\times V} \tag{2.2}$$

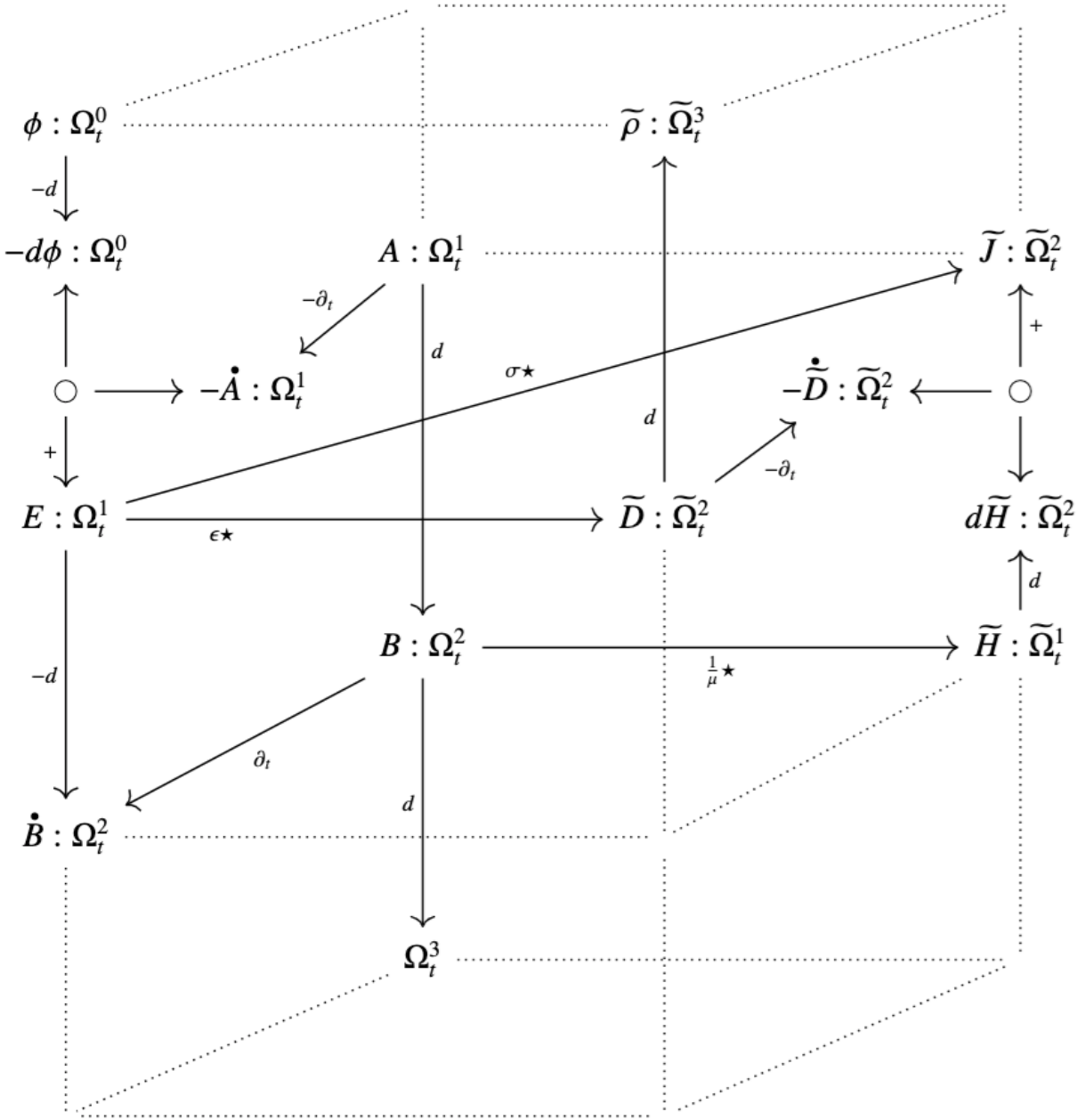
Example 6.5 (Maxwell's house). Let M be a three-dimensional Riemannian manifold. Phrased in exterior calculus and with units such that the speed of light is 1, Maxwell's equations on the spatial domain M are

$$dE = -\partial_t B \qquad E = -d\phi - \partial_t A$$
$$dB = 0 \qquad B = dA.$$
$$\delta E = \rho$$
$$\delta B = \partial_t E + J,$$

$$\widetilde{D} := \epsilon \star E \qquad \text{and} \qquad \widetilde{H} := \frac{1}{\mu} \star B,$$

$$d\widetilde{D} = \widetilde{\rho}$$
$$d\widetilde{H} = \partial_t \widetilde{D} + \widetilde{J},$$
$$\widetilde{J} = \sigma \star E,$$

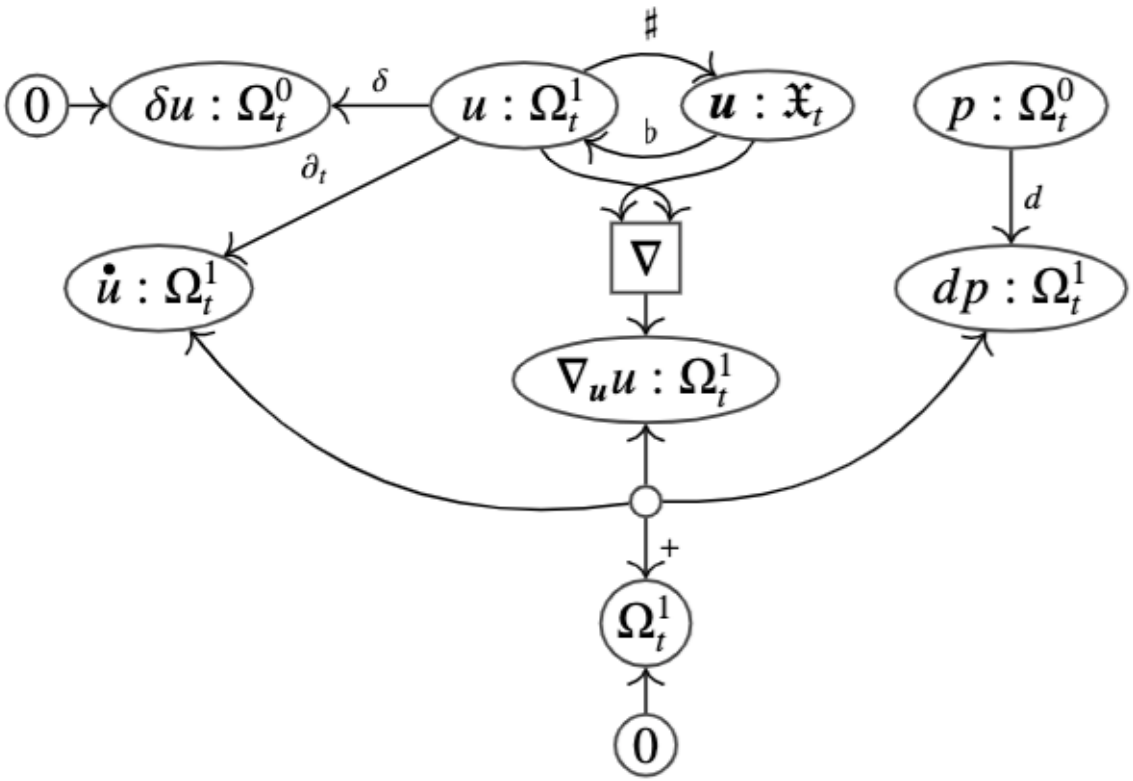
Maxwell's house



Incompressible Euler equations

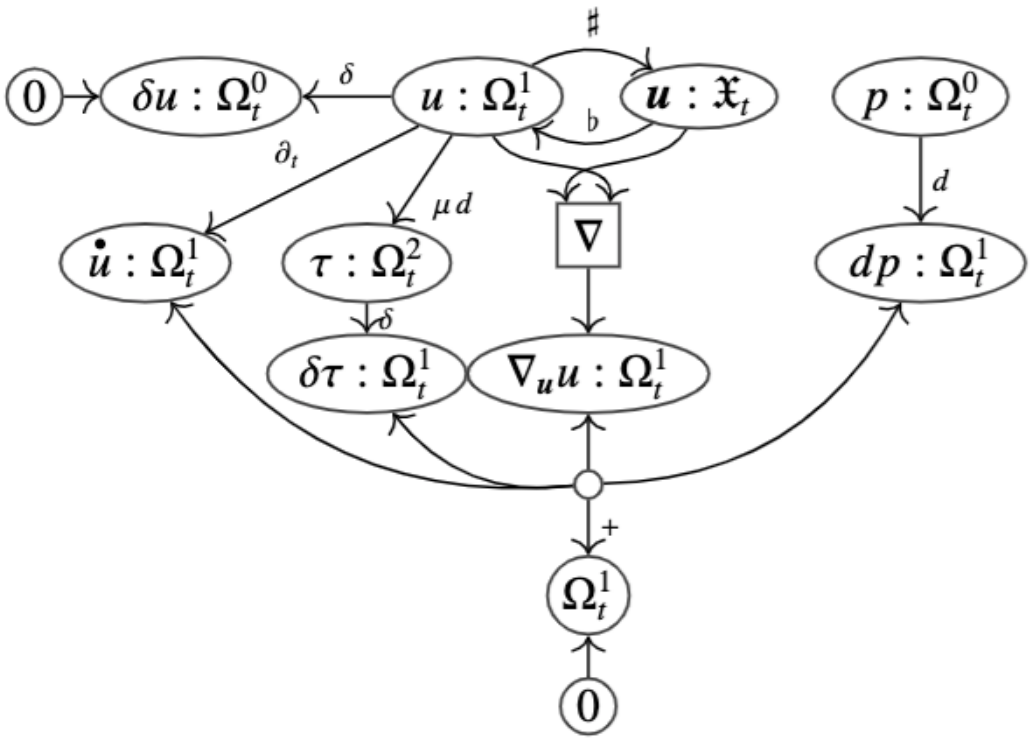
$$\partial_t u + \nabla_u u + dp = 0$$
$$\delta u = 0,$$

$$\partial_t u + \mathcal{L}_u u - \frac{1}{2} d\|\boldsymbol{u}\|^2 + dp = 0.$$



Incompressible Navier-Stokes equations

$$\partial_t u + \nabla_u u + \mu\, \delta du + dp = 0$$
$$\delta u = 0,$$



Appendix

How to Generate Code from Graph

Decapodes.jl compiler

```
module Decapodes
using ACSets
using DiagrammaticEquations
using DiagrammaticEquations.Deca
using MLStyle
export
gensim, evalsim, compile, compile_env,
default_dec_matrix_generate, default_dec_cu_matrix_generate,
default_dec_generate,CPUBackend, CUDABackend, CPUTarget,
CUDATarget

include("operators.jl")
include("simulation.jl")

# documentation
include("canon/Canon.jl")
end
```

operators.jl

```
using Base.Iterators
using CombinatorialSpaces
import CombinatorialSpaces.DiscreteExteriorCalculus:DiscreteHodge
using Krylov
using LinearAlgebra
using SparseArrays


function default_dec_cu_matrix_generate() end;
function default_dec_matrix_generate(sd::HasDeltaSet,
  my_symbol::Symbol, hodge::DiscreteHodge)
function dec_mat_hodge(k::Int, sd::HasDeltaSet,
  hodge::DiscreteHodge)
function dec_mat_inverse_hodge(k::Int, sd::HasDeltaSet,
  hodge::DiscreteHodge)
# Special case for inverse hodge for DualForm1 to Form1
function dec_pair_inv_hodge(::Type{Val{1}},
  sd::AbstractDeltaDualComplex2D, ::GeometricHodge)
```

function dec_pair_inv_hodge(::Type{Val{1}}, sd::HasDeltaSet, ::DiagonalHodge)

function dec_mat_differential(k::Int, sd::HasDeltaSet)

function dec_mat_dual_differential(k::Int, sd::HasDeltaSet)

function dec_pair_wedge_product(::Type{Tuple{k,0}}, sd::HasDeltaSet) where {k}

function dec_pair_wedge_product(::Type{Tuple{0,k}}, sd::HasDeltaSet) where {k}

function dec_pair_wedge_product(::Type{Tuple{0,0}}, sd::HasDeltaSet)

function dec_#_d(sd::HasDeltaSet2D)

function dec_♭(sd::HasDeltaSet2D)

function dec_avg$_{01}$(sd::HasDeltaSet)

function default_dec_generate(sd::HasDeltaSet, my_symbol::Symbol, hodge::DiscreteHodge=GeometricHodge())

function open_operators(d::SummationDecapode; dimension::Int=2)

function open_operators!(d::SummationDecapode; dimension::Int=2)

```
op2_remove_stack = Vector{Int}(
function add_Inter_Prod!(d::SummationDecapode, proj1_Inter::Int,
  proj2_Inter::Int, res_Inter::Int)
function add_Inter_Prod_2D!(::Type{Val{2}},
  d::SummationDecapode, proj1_Inter::Int, proj2_Inter::Int,
  res_Inter::Int)
function add_Lie_1D!(::Type{Val{0}}, d::SummationDecapode,
  proj1_Lie::Int, proj2_Lie::Int, res_Lie::Int)
function add_Lie_2D!(::Type{Val{0}}, d::SummationDecapode,
  proj1_Lie::Int, proj2_Lie::Int, res_Lie::Int)
function add_Lie_2D!(::Type{Val{1}}, d::SummationDecapode,
  proj1_Lie::Int, proj2_Lie::Int, res_Lie::Int)
function add_Lie_2D!(::Type{Val{2}}, d::SummationDecapode,
  proj1_Lie::Int, proj2_Lie::Int, res_Lie::Int)
function add_Codiff!(d::SummationDecapode, src_Codiff::Int,
 tgt_Codiff::Int)
function add_De_Rham_1D!(::Type{Val{0}},
  d::SummationDecapode, src_De_Rham::Int, tgt_De_Rham::Int)
```

```
function add_De_Rham_1D!(::Type{Val{1}},
  d::SummationDecapode, src_De_Rham::Int, tgt_De_Rham::Int)
function add_De_Rham_2D!(::Type{Val{0}},
  d::SummationDecapode, src_De_Rham::Int, tgt_De_Rham::Int)
function add_De_Rham_2D!(::Type{Val{1}},
  d::SummationDecapode, src_De_Rham::Int, tgt_De_Rham::Int)
function add_De_Rham_2D!(::Type{Val{2}},
  d::SummationDecapode, src_De_Rham::Int, tgt_De_Rham::Int)
```

simulation.jl

```
using CombinatorialSpaces
using ComponentArrays
using LinearAlgebra
using MLStyle
using PreallocationTools

const GENSIM_INPLACE_STUB = Symbol("GenSim-M")
const NO_STUB_RETURN = Symbol("NOSTUB")

abstract type AbstractGenerationTarget end

abstract type CPUBackend <: AbstractGenerationTarget end
abstract type CUDABackend <: AbstractGenerationTarget end

struct CPUTarget <: CPUBackend end
struct CUDATarget <: CUDABackend end
```

```
# TODO: Make it so AbstractCall code terminates into an error, not into default code
abstract type AbstractCall end

struct InvalidCallException <: Exception end
Base.showerror(io::IO, e::InvalidCallException) = print(io,
  "Compiler call being made is not a valid one")

# A catch all if an AbstractCall's child doesn't define `Base.Expr`
Base.Expr(::AbstractCall) = throw(InvalidCallException)

struct UnaryCall <: AbstractCall
  operator::Union{Symbol, Expr}
  equality::Symbol
  input::Symbol
  output::Symbol
end
```

```
# ! WARNING: Do not pass this an inplace function without setting equality to :.=
Base.Expr(c::UnaryCall)

struct BinaryCall <: AbstractCall
  operator::Union{Symbol, Expr}
  equality::Symbol
  input1::Symbol
  input2::Symbol
  output::Symbol
end

# ! WARNING: Do not pass this an inplace function without setting equality to :.=, vice versa
Base.Expr(c::BinaryCall) = begin
```

```
struct VarargsCall <: AbstractCall
  operator::Union{Symbol, Expr}
  equality::Symbol
  inputs::Vector{Symbol}
  output::Symbol
end

Base.Expr(c::VarargsCall) = begin
  return Expr(c.equality, c.output, Expr(:call, c.operator,
c.inputs...))
end

struct AllocVecCall <: AbstractCall
  name::Symbol
  form::Symbol
  dimension::Int
  T::DataType
  code_target::AbstractGenerationTarget
end
```

```
struct AllocVecCallException <: Exception
  c::AllocVecCall
end
# TODO: Should maybe have default CPU generation be Vector with PreallocTools being opt-in
"""
    hook_AVC_caching(c::AllocVecCall, resolved_form::Symbol, ::CPUBackend)

This hook can be overridden to change the way in which vectors can be preallocated for use by in-place functions.
The AllocVecCall stores the `name` of the vector, the `form` type, the `dimension` of the simulation, the `T` which is the
datatype of the vector, and the `code_target` which is used by multiple dispatch to select a hook.
An example overloaded hook signature would be `hook_AVC_caching(c::AllocVecCall, resolved_form::Symbol, ::UserTarget)`
"""
```

```
function hook_AVC_caching(c::AllocVecCall,
resolved_form::Symbol, ::CPUBackend)
  :($(Symbol(:__,c.name)) =
Decapodes.FixedSizeDiffCache(Vector{$(c.T)}(undef, nparts(mesh,
$(QuoteNode(resolved_form))))))
end

# TODO: Allow user to overload these hooks with user-defined
code_target
function hook_AVC_caching(c::AllocVecCall,
resolved_form::Symbol, ::CUDABackend)
  :($(c.name) = CuVector{$(c.T)}(undef, nparts(mesh,
$(QuoteNode(resolved_form)))))
end

"""
    compile_var(alloc_vectors::Vector{AllocVecCall})
```

This creates the vector allocations that will be used by the simulation body for in-place operations.
"""
function compile_var(alloc_vectors::Vector{AllocVecCall})
  return quote $(Expr.(alloc_vectors)...) end
end

TODO: Why do we need a QuoteNode for this?
"""
    hook_STC_settvar(src_name::Symbol, tgt_name::Symbol, ::Union{CPUBackend, CUDABackend})

This hook is meant to control how data is set into the tangent variables after a simulation function
execution. It expects the `state_name`, the name of the original state variable, the `tgt_name` which
is the name of the variable whose data will be stored and a code target.
"""