DNP アニーリング・ソフトウェア

v1.0.4

ユーザー・マニュアル

（第 1.0 版）

大日本印刷株式会社

目次

1　はじめに

1.1　　本書の適用範囲

本書は、大日本印刷株式会社が提供する「DNP アニーリング・ソフトウェア v1.0.4」（以下、本ソフトウェア）を利用するためのユーザー・マニュアルである。
※本ソフトウェアの旧バージョンをすでに利用中の場合は該当するバージョンに対応するマニュアルを参照すること。

1.2　　本ソフトウェアの概要

工場の生産計画や人員計画の最適化、物流や運送便の経路の最適化、複数エリアでの最適なセールス経路の決定など、社会生活や企業活動で直面する多様な組合せ最適化問題を解くための技術として数理最適化技術があるが、規模の大きな問題を解こうとすると現実的な計算時間で解けない場合があるという課題があった。それに対して、近年、超高速で組合せ最適化問題を解くことができると言われている量子アニーリングマシンやそれを模した専用チップを利用した疑似量子アニーリングマシンといった製品・サービスが登場してきた。しかし、特別なハードウェアを前提としているため利用に際しては高額な投資が必要という問題があった。

本ソフトウェアは、特別なハードウェアを前提とせず GPU を搭載した PC があれば、最適解を高速で求めることができるソフトウェアである。本ソフトウェアは、アニーリング法による組合せ最適化問題を汎用的に解くことができるライブラリ（Python パッケージ）であり、Python プログラムの中から適切な機能を呼び出して利用する。

実際の問題を解く際には、入力条件に応じてアニーリング計算に合わせたデータセットを用意して、本ソフトウェアに入力する。最適化計算の結果はバイナリ系列で得られるため、人間が目で見てわかる情報に解釈・変換する必要がある。従って、本ソフトウェアによる処理の前後に、解きたい問題に応じた処理をプログラム実装する必要がある。本ソフトウェアの使用にあたっては、アニーリングを用いた組合せ最適化問題の定式化に関する知識に加え、Python 言語でのプログラミングスキルを前提とする。

1.3　　動作環境

本ソフトウェアは、下記環境で動作する。

OS	・Ubuntu 24.04 LTS
GPU	・NVIDIA CUDA 12.6
Python	・Python v3.12
Python パッケージ	・numpy 1.26

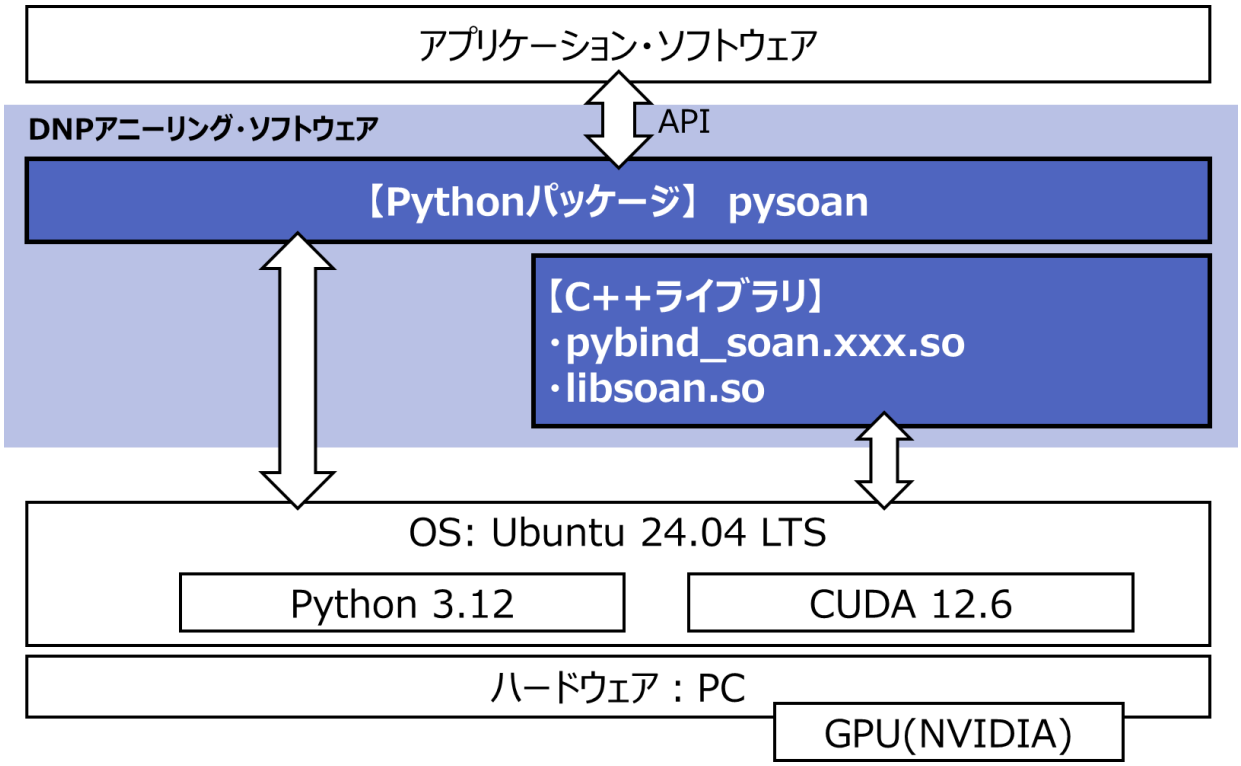
※下記スペック相当以上のハードウェアを推奨する。

【メモリ】16GB

【GPU】NVIDIA GeForce RTX 3050（GPU メモリ：8GB）

1.4　　システム構成

本ソフトウェアが動作するシステム構成の概略図を示す。



2　使用方法

2.1　　インストール

ライセンス購入契約後に提供されたライセンスファイル’license.txt’を~/.pysoan ディレクトリにコピーする。

```
$ mkdir ~/.pysoan
$ cp license.txt ~/.pysoan
```

同様に提供された pysoan_dist_1.0.4.tgz を適当なディレクトリで展開する（以下では、ユーザーのホームディレクトリに展開する想定とする）。

```
$ cd
$ tar xvfz pysoan_dist_1.0.4.tgz
```

展開されたディレクトリに移動する。

```
$ cd pysoan_dist_1.0.4
```

pysoan 本体をインストールするため、以下のコマンドを実行する。

```
$ pip install -e .
```

これでインストールは完了である（インストール後も展開したディレクトリの内容は消さないこと）。正常にインストールされたかどうかは、sample ディレクトリにあるサンプルコード basic_model.py を以下のように実行してすれば確認できる。

```
$ cd sample
$ python basic_model.py
* qubits =  [0, 0, 0, 0]
* energy =  0.0
* time   =  0.00074
```

2.2　　アンインストール

以下のように、pysoan をアンインストールして、プログラム本体を削除し、ライセンスファイルを削除する。

```
$ pip uninstall pysoan
$ cd
$ rm -rf pysoan_dist_1.0.4
$ rm ~/.pysoan/license.txt
```

2.3　機能一覧

●モデル設定：解きたい最適化問題を記述するデータ系列を設定
●パラメータ設定：アニーリング計算を制御するためのパラメータを設定
●求解：上記設定に基づきアニーリング計算を実行し、結果データを取得

カテゴリ No	カテゴリ	機能 No	機能名	機能概要
001	モデル設定	F001-001	係数リスト指定	イジングモデル（バイナリ変数が-1/+1）または QUBO モデル（バイナリ変数が 0/1）における目的関数(*1)を、モデルタイプ、サイズ、定数項および一次係数/二次係数のリストから成る独自の辞書形式にして設定する。
		F001-002	QUBO 行列リスト指定	QUBO モデルにおける目的関数を、サイズおよび QUBO 行列に対応した二次元リストから成る独自の辞書形式にして設定する。
		F001-003	係数辞書データ指定	イジングモデルまたは QUBO モデルの目的関数に含まれる非ゼロ係数からなるデータを辞書形式にして設定する。
		F001-004	PyQUBO(*2)データ指定	PyQUBO によって作成された QUBO 記述データを設定する。
002	パラメータ設定	F002-001	自動モード設定	True にすると、F002-004～010 のアニーリング制御パラメータを問題に応じて内部で自動設定する（F002-004～010 の設定は不要となる）。
		F002-002	アニーリングアルゴリズム設定	シミュレーテッドアニーリング（SA）、パラレルテンパリング（PT）、量子モンテカルロ（QMC）のどれかを設定する。または ALL を設定すると 3 手法を実行して一番良い解を取得する。
		F002-003	変数セット数設定	バイナリ変数セットを内部的に何セット用意して計算するかを設定する。
		F002-004	アニーリングステップ数設定	温度や磁場を変化させながら実行するアニーリングのステップ数を設定する。
		F002-005	繰り返し数設定	アニーリング計算を何回繰り返すかを設定する（繰り返しごとに異なる結果から一番良いものを取得する）。
		F002-006	動作環境設定	CPU で実行するか GPU で実行するか選択する。

		F002-007	アニーリングスケジュールのマニュアル設定（SA/QMC のみ）	アニーリングステップごとに温度や磁場（QMC のみ）をどう変化させるかを任意に設定する。
		F002-008	アニーリングスケジュールの変化曲線タイプ設定（SA/QMC のみ）	アニーリングスケジュールを任意設定しない場合、スケジューリングの変化曲線のタイプを設定する（linear（一次関数）または power（べき関数）または exponent（指数関数））。
		F002-009	アニーリングスケジュールの初期温度設定（SA のみ）	アニーリングスケジュールの変化曲線タイプを設定し、アルゴリズムを SA とした場合、温度の初期値を設定する。
		F002-010	アニーリングスケジュールの初期磁場設定（QMC のみ）	アニーリングスケジュールの変化曲線タイプを設定し、アルゴリズムを QMC とした場合、磁場の初期値を設定する。
		F002-011	固定温度設定（QMC のみ）	アニーリングスケジュールの変化曲線タイプを設定し、アルゴリズムを QMC とした場合、一定に保つ温度の値を設定する。
		F002-012	アニーリングスケジュールの変化指数設定（SA/QMC のみ）	アニーリングスケジュールの変化曲線タイプを設定した場合、曲線の緩急を表す変化指数を設定する。
		F002-013	並行温度最大値設定（PT のみ）	並行温度系列の最大温度を設定する。
		F002-014	並行温度指数設定（PT のみ）	並行温度系列の温度間隔を制御する指数を設定する。
003	求解	F003-001	アニーリング実行	アニーリング処理を実行して結果データを取得する。
		F003-002	バイナリ変数系列取得	結果データから、目的関数を最小化するバイナリ変数系列を取得する。
		F003-003	エネルギー値取得	結果データから、目的関数の最小値（エネルギー値）を取得する。
		F003-004	計算時間取得	結果データから、アニーリング処理にかかった計算時間を取得する。

2.4　プログラムの流れ

本ソフトウェアを使用する基本的な流れは、

① Python モジュールのインポート
② ソルバー初期化
③ モデル設定
④ パラメータ設定
⑤ 求解
⑥ ソルバー解放

である。各ステップにおいて設定できるオプション等の詳細について、以下に説明する。

① Python モジュールのインポート

DNP アニーリング・ソフトウェアの機能を実行するためのクラス’SoanSolver’を import する。

```
from pysoan.solver.soan_solver import SoanSolver
```

さらに、アニーリング結果のデータからバイナリ系列、目的関数の最小値（エネルギー値）、計算時間を取得するための関数’GetQubits’,’GetEnergy’,’GetTime’を import する。

```
from pysoan.util.result import GetQubits,GetEnergy,GetTime
```

② ソルバー初期化

‘SoanSolver’クラスのコンストラクタを呼び出してインスタンスを生成し、適当な変数（例えば solver）に格納する。以降、この solver に対して各種メソッドを呼び出すことで、一連の処理が実行される。

```
solver = SoanSolver()
```

③ モデル設定

解きたい問題の目的関数が以下のように与えられた時、係数や定数項の値をあるデータ構造として設定することを「モデル設定」と呼んでいる。

$$H = \sum_{i=0}^{N-1}\sum_{j>i}^{N-1} J_{ij}x_i x_j + \sum_{i=0}^{N} h_i\, x_i + C \qquad (1)$$

ここで、x_iは N 個のバイナリ変数値（-1/+1 または 0/1 の値をとる。前者を「イジングモデル」、後者を「QUBO モデル」と呼ぶ）を表しており、係数J_{ij}, h_i, Cは各々実数値である。

モデル設定には「(a)係数リストによる方法」と「(b)QUBO 行列リストによる方法」「(c)係数辞書データによる方法」「(d)PyQUBO データによる方法」の４種類の方法がある。以下、順に説明する。

(a) 係数リストによる方法

簡単な例を示す。

```
model = {
    ‘model_type’ : ‘qubo’,
    ‘size’ : 16,
    ‘const’ : 32.0,
    ‘term_1’ : [0, 1, ..., 15],
    ‘coef_1’ : [-2.0, -2.0, ..., -2.0],
    ‘term_2’ : [[0,1], [0,2], [0,3],..., [14,15]],
    ‘coef_2’ : [4.0, 6.0, 8.0, ..., 24.0]
}

solver.set_model(model=model)
```

‘model_type’,’ size’,’ const’,’ term_1’,’ coef_1’,’ term_2’,’ coef_2’ をキーとした辞書データを用意して適当な変数として格納する（例えば、model）。これを set_model メソッドの’model’オプションに引数として与えることで解きたい問題に対応したモデルが設定される。各キーと値の型およびその説明は以下の通りである。

キー（文字列）	値の型	値の説明
model_type	string	「イジングモデル」(変数値が-1/+1)の場合文字列’ising’、「QUBO モデル」の場合文字列’qubo’を指定する[任意：defaultは’ising’]。
size(*3)	int	変数の数を指定する[必須]。最大値は 40,000 である。
const	float	目的関数の定数項の値を指定する[必須]。
term_1	list(int)	目的関数の 1 次の項の中で係数値が 0 以外のものの変数番号のリストを指定する[任意:default は[0,1,…,size-1]]。
coef_1	list(float)	上記変数番号に対応した係数値を上記リストの順番に従い配置したリストを指定する[任意:default は 0.0 が size 個並んだリスト]。 *※term_1 と coef_1 の要素数は一致していなければならない。*
term_2	list([int,int])	目的関数の 2 次の項の中で係数値が 0 以外のものの変数番号の組（2 要素からなるリスト)のリストを指定する[任意:default はすべての変数番号の組み合わせ]。 *※同じ変数番号を組にしたものはあってはならない（例：[…, [3,3], …]は NG）。また、順番を入れ替えた２つの組もあってはならない（例：[…, [2,4], …, [4,2], …]は NG）。*
coef_2	list(float)	上記変数番号の組に対応した係数値を上記リストの順番に従い配備したリストを指定する[任意:default は 0.0 が size 個並んだリスト]。 *※term_2 と coef_2 の要素数は一致していなければならない。*

(b) QUBO 行列リストによる方法

QUBO モデルでモデル設定する場合、QUBO 行列リストによる方法を使うこともできる。簡単な例を示す。

```
qmat = {'const':3.0,
        'mat':[[1.0, 2.0, 2.0, 2.0],
               [0.0, 1.0, 2.0, 2.0],
               [0.0, 0.0, 1.0, 2.0],
               [0.0, 0.0, 0.0, 1.0]]}
solver.set_model(qmat=qmat)
```

‘const’,’ mat’ をキーとした辞書データを用意して適当な変数として格納する（例えば、qmat）。これを set_model メソッドの’qmat’オプションに引数として与えることで解きたい問題に対応したモデルが設定される。各キーと値の型およびその説明は以下の通りである。

キー（文字列）	値の型	値の説明
const	float	目的関数の定数項の値を指定する。[任意:default は 0.0]
mat	list(list(float))	リストのリストで与えられた上三角正方行列（左下三角領域がすべて 0 であるような正方行列）を指定する[必須]。その際、目的関数の 1 次の係数を対角成分に配置し、2 次の項に対する係数値をそれ以外の領域に配置する。行列要素をM_{ij}とすると、$h_i = M_{ii}$, $J_{if} = M_{ij}$ のようにする。

(c) 係数辞書データによる方法

イジングモデルや QUBO モデルの二次係数および一次係数は、変数の数に相当する次元を持つ行列で表現することができる。前段(b)のキー’mat’の値に指定していたものは QUBO モデルの場合の行列表現である。QUBO モデルの一次係数を対角要素に並べ、二次係数を非対角の右上三角領域に配置したものを二次元リスト（リストのリスト）として表現していた。イジングモデルの場合も同様に行列表現することができる。しかし、モデル化を実際に行うと、行列要素のほとんどがゼロになる場合も少なくないため、行列表現によるモデル化はメモリ消費の観点であまり良いデータ構造ではない。そこで、ゼロでない要素のみに注目して、i 行、j 列にある非ゼロの要素の値が v（≠0）であったときに(i,j)をキー、値を v にした辞書データで表現することが、よく行われている。DNP アニーリング・ソフトウェアでも、この形式を入力できるようになっている。

QUBO モデルの場合の例を示す。

```
Q = {(0,0):1.0, (1,1):1.0, (2,2):1.0, (3,3):1.0,
     (0,1):2.0, (0,2):2.0, (0,3):2.0, (1,2):2.0, (1,3):2.0, (2,3):2.0}
solver.set_model(Q=Q)
```

イジングモデルの場合は、以下のように、二次係数、一次係数を各々別に用意する。

```
J = {(0,1):2.0, (0,2):2.0, (0,3):2.0, (1,2):2.0, (1,3):2.0, (2,3):2.0}
h = {0:1.0, 1:1.0, 2:1.0, 3:1.0}
solver.set_model(J=J, h=h)
```

上記の例では、変数のアドレスは整数で表現されているが、以下のように文字列で記述することも可能である。

QUBO モデルの場合、

```
Q = {('a','a'):1.0, ('b','b'):1.0, ('c','c'):1.0, ('d','d'):1.0,
('a','b'):2.0,('a','c'):2.0,('a','d'):2.0,('b','c'):2.0, ('b','d'):2.0, ('c','d'):2.0}
solver.set_model(Q=Q)
```

イジングモデルの場合、

```
J = {('a','b'):2.0, ('a','c'):2.0, ('a','d'):2.0, ('b','c'):2.0,
 ('b','d'):2.0, ('c','d'):2.0}
h = {'a':1.0, 'b':1.0, 'c':1.0, 'd':1.0}
solver.set_model(J=J, h=h)
```

のようにも記述できる。

このように記述して求解した場合、GetQubits 関数で得られる結果はリストではなく、文字列をキーとした辞書になることに注意を要する（詳細は後述）。また、各モデルに含まれる定数項を無視した形式になっているため、GetEnergy 関数で得られるエネルギー値は、本来のエネルギー値ではないことにも注意を要する。本来のエネルギー値を得たい場合は、別途（計算し）保持しておいた定数項の値を GetEnergy 関数の出力値に加算する必要がある。

(d) PyQUBO データによる方法

組合せ最適化問題をアニーリング法で解く場合の定式化には、通常Σや添え字付きの変数からなる目的関数の数式を展開して二次係数や一次係数や定数項に対応した数式表現を得た上で、その数式表現に具体的な入力データを当てはめて二次係数や一次係数や定数項の具体的な値を出力するプログラムを作成することが行われている。つまり、数式表現から二次係数や一次係数や定数項の値を得るために、数式展開の手作業および前処理のプログラミングという面倒な作業が必要となる。PyQUBO はその作業負荷を削減させるツールである。目的関数の数式表現をそのままプログラムとして記載するだけで、必要な係数リストを出力してくれる。DNP アニーリング・ソフトウェアでは、PyQUBO が出力した形式も入力できるようになっている。

簡単な例を示す。

```
# --- PyQUBO でモデル化 ---
x = Array.create('x', shape=(4), vartype='BINARY') # 変数: x
H = sum(x[i] for i in range(4))**2                 # 目的関数: H = (Σi xi)^2
H_compiled = H.compile()
qubo, offset = H_compiled.to_qubo()
# ---

solver.set_model(pyqubo=(qubo, offset))
```

このように記述して求解した場合、GetQubits 関数で得られる結果は Array.create 時に設定した変数文字列をどう設定したかに関わらず、単なるリストになる。上の例では[0,0,0,0]という解が得られるはずである。リストの 0 番目の要素が変数 x[0]の値、1 番目の要素が変数 x[1]の値…と解釈される。また、PyQUBO では多次元変数も扱うことができる。その場合、GetQubits 関数の出力は多次元リストとなる（詳細は後述）。

④ パラメータ設定

モデルが設定できたら、次に set_param メソッドを使ってアニーリング処理を制御するための各種パラメータを設定する。

以下にいくつかの具体的な設定例を示す。まず、量子モンテカルロ法を用いて、磁場および温度の変化量をカスタム設定した例を示す。

```
anneal_time = 100
gamma = [100.0-i*10.0 for i in range(anneal_time)]
temper = [0.05 for i in range(anneal_time)]
schedule ={'algorithm':'QMC','anneal_time':anneal_time,'temper':temper,'gamma':gamma}
solver.set_param(schedule=schedule)
```

次の例は、アニーリングスケジューリングの変化曲線をカスタム設定ではなく曲線パターン（下の例は’一次関数’）で設定した例である。

```
solver.set_param(algorithm='SA',repetition=20,schedule_type='Linear',trotter=8,
                 qmc_temper=100.0, decrease_factor=2.0,
                 anneal_time=100)
```

次の例は、各種パラメータ設定を自動で行うようにした例である。

```
solver.set_param(auto=True, algorithm='SA')
```

‘algorithm’オプションに’ALL’を指定するとすべてのアニーリング手法で計算して、一番良い解を取得することができる。

```
solver.set_param(auto=True, algorithm='ALL')
```

set_param メソッドのオプションと型、およびその説明を以下の表にまとめる。

●基本パラメータ

オプション	型	説明
auto	bool	アニーリング制御パラメータを自動設定するか否かを表す[任意:default は False]。 *※True に設定された場合、基本設定以外のパラメータは無視される（パラメータは設定されたモデルに応じて内部で自動設定される）。*
algorithm	string	アニーリングのアルゴリズムを指定する[任意:default は’ALL’]。 ・シミュレーテッドアニーリング　⇒’SA’ ・量子モンテカルロ　⇒’QMC’ ・パラレルテンパリング　⇒’PT’ ・すべて実行（最良解取得）　⇒’ALL’ *※auto=False の場合、’ALL’を設定することはできません。*

trotter(*4)	int	バイナリ変数セットを内部的に何セット用意して計算するかを設定する[任意:default は 8]。
anneal_time(*4)	int	アニーリングのステップ数を設定する[任意:default は 100]。
repetition(*4)	int	アニーリング計算を何回繰り返すかを整数値で設定する（繰り返しごとに得られた異なる結果から一番良いものを取得する）[任意:default は 1]。
proc	string	CPU で実行するか GPU で実行するか選択する[任意:default は’CPU’]。

●シミュレーテッドアニーリング、量子モンテカルロのためのパラメータ

オプション	型	説明
schedule	dict	‘algorithm’,’anneal_time’,’temper’,’gamma’ をキーとした辞書で指定する[任意:default は None]。’algorithm’はシミュレーテッドアニーリング‘SA’または量子モンテカルロ‘QMC’のうちのどちらかを指定する。’anneal_time’はアニーリングのステップ数を整数値で指定する。’temper’と’gamma’は各々温度および磁場の変化をリストで指定する（’SA’の場合’gamma’は不要である（原理上）。指定されていたとしても無視される）。’temper’および’gamma’のリストサイズは’anneal_time’と一致していなければならない。また、基本設定の’annela_time’とこの辞書中の’anneal_time’は一致していなければならない。’algorithm’についても同様である。以下に指定例を示す。 例：{'algorithm':'QMC','anneal_time':100, 'temper':[0.01,0.01,...], 'gamma':[100.0,99.0,...]}
※‘schedule’が指定されている場合、以下のオプション（’schedule_type’,’sa_temper’,’qmc_temper’,’qmc_gamma’,’decrease_factor’）は指定されていたとしても無視される。‘schedule’が指定されていない場合のみ、有効になる。		
schedule_type	string	アニーリングスケジュールの変化曲線のタイプを指定する[任意:default は’Power’]。 ・一次関数　⇒’Linear’ ・べき関数　⇒’Power’ ・指数関数　⇒’Expo’ パラレルテンパリングの場合この指定は無視される。
sa_temper	float	シミュレーテッドアニーリングの温度の初期値を設定する[任意:default は 100.0]。
qmc_gamma	float	量子モンテカルロの磁場の初期値を設定する[任意:default は 10.0]。
qmc_temper	float	量子モンテカルロで固定温度動作させる際の温度の値を設定する[任意:default は 0.01]。

decrease_factor	float	‘schedule_type’が設定されている場合、各々の変化量 r を指定するための因子を設定する[任意:default は 0.9]。アニーリングステップを t としたとき、 ・’Linear’⇒ -rt+[初期値]で変化する直線を生成する。 ・’Power’ ⇒ [初期値]×[(t+1)の-r 乗]で変化する曲線を生成する。 ・’Expo’ ⇒ [初期値]×[2 の-rt 乗]の曲線を生成する。

●パラレルテンパリングのためのパラメータ

オプション	型	説明
pt_temper	float	並行温度の最大値を指定する[任意:default は 100]。
pt_factor	float	基本設定の’trotter’で指定された数の並行温度を設定するが、その温度間隔を制御する指数を指定する[任意:default は 0.3]。

⑤ 求解

モデル設定とパラメータ設定が完了したら、アニーリング処理を実行するため、’solve’メソッドを呼び出す(*5)。返り値を適当な変数（例えば、result）で受け取る。

```
result = solver.solve()
```

この result に結果のデータが格納されているため、’GetQubits’関数で、

```
qubits = GetQubits(result)
```

のように目的関数を最小化するバイナリ変数系列’qubits’（整数値のリストまたは辞書）を取得できる。モデル設定をどのような方法で行ったかによってそのデータ形式が異なるため、以下に整理する。

(a) 係数リストによる方法でモデル化を行った場合

【QUBO モデル】モデルが想定する変数番号順に変数の値が並んでいるリスト

例）[0,1,0,1,1,0]

【イジングモデル】モデルが想定する変数番号順に変数の値が並んでいるリスト

例）[-1,1,-1,1,1,-1]

(b) QUBO 行列リストによる方法

【QUBO モデル】モデルが想定する変数番号順に変数の値が並んでいるリスト

例）[0,1,0,1,1,0]

【イジングモデル】モデル化は不可

(c) QUBO 行列リストによる方法

【QUBO モデル（キー：整数値）】モデルが想定する変数番号順に変数の値が並んでいるリスト

例）[0,1,0,1,1,0]

【QUBO モデル（キー：文字列）】変数文字列を「キー」、その変数の値を「値」にした辞書

例）{‘a’:0, ‘b’:1, ‘c’:0, ‘d’:1, ‘e’:1, ‘f’:0}

【イジングモデル（キー：整数値）】モデルが想定する変数番号順に変数の値が並んでいるリスト

例）[0,1,0,1,1,0]

【イジングモデル（キー：文字列）】変数文字列を「キー」、その変数の値を「値」にした辞書

例）{‘a’:-1, ‘b’:1, ‘c’:-1, ‘d’:1, ‘e’:1, ‘f’:-1}

(d) PyQUBO データによる方法

【QUBO モデル（一次元変数）】モデルが想定する変数番号順に変数の値が並んでいるリスト

例）[0,1,0,1,1,0]

【QUBO モデル（多次元変数）】モデルが想定する変数番号順に変数の値が並んでいるリスト

例）[[0,1,0],[1,1,0]]

※問題をイジングモデルで定式化した場合は、PyQUBO の機能で QUBO 形式データとして出力する必要がある。DNP アニーリング・ソフトウェアで求解した解は QUBO 形式となるため、解の変数値に含まれる「0」を「-1」に読み替えることで、イジングモデルとしての解を得ることができる。

得られた解に対する目的関数の値は、’GetEnergy’関数で、

```
energy = GetEnergy(result)
```

のように取得できる。アニーリング法は基本的に内部でランダムに解を探索する方法なので、最低エネルギー付近の解が複数ある場合は、実行のたびに結果が異なる場合があることに注意を要する。

アニーリングの計算時間(sec)は’GetTime’関数で、

```
ptime = GetTime(result)
```

のように取得することができる。

⑥ ソルバー解放

```
solver.free()
```

計算結果が得られたら、最後に計算に使用していたメモリをすべて解放するため、’free’メソッドを呼び出す。

2.5　　簡単な使用例

バイナリ変数$\{x_i\}$ $(i = 0,1,2,3)$に対して、定義された以下の目的関数を最小化する問題を解くプログラム例を示す。

目的関数：

$$H = 2\sum_{i=0}^{3}\sum_{j>i}^{3} x_i x_j + \sum_{i=0}^{3} x_i \qquad (2)$$

(a) 係数リストによりモデル設定した例

```
# example_a.py

from pysoan.solver.soan_solver import SoanSolver
from pysoan.util.result import GetQubits,GetEnergy,GetTime

# 変数サイズ
size = 4

# 定数項
const = 0.0

# 1 次の項
term_1 = [0, 1, 2, 3]           # ← 変数番号
coef_1 = [1.0, 1.0, 1.0, 1.0]  # ← 1 次の項の係数値

# 2 次の項
term_2 = [[0,1], [0,2], [0,3], [1,2], [1,3], [2,3]]  # ← 変数番号の組
coef_2 = [2.0, 2.0, 2.0, 2.0, 2.0, 2.0]              # ← 2 次の項の係数値

# モデル
model = {'model_type': 'qubo', 'size': size, 'const': const,
         'term_1': term_1, 'coef_1': coef_1,
         'term_2': term_2, 'coef_2': coef_2}

# ソルバーを生成
solver = SoanSolver()

# ソルバーにモデルをセット
solver.set_model(model=model)

# ソルバーにパラメータをセット
solver.set_param(auto=True, algorithm='ALL', proc=' GPU')

# 求解
result = solver.solve()

# 結果を表示
print( '* qubits =  ', GetQubits(result))
print( '* energy =  ', GetEnergy(result))
print( '* time   =  ', GetTime(result))

# ソルバーを解放
solver.free()
```

【実行結果】

```
$ python example_1.py
* qubits =  [0, 0, 0, 0]
* energy =  0.0
* time   =  0.033858
```

(b) QUBO 行列リストによりモデル設定した例

```
# example_b.py

from pysoan.solver.soan_solver import SoanSolver
from pysoan.util.result import GetQubits,GetEnergy,GetTime

# QUBO 行列
qmat = {'const':0.0,
        'mat':[[1.0, 2.0, 2.0, 2.0],
               [0.0, 1.0, 2.0, 2.0],
               [0.0, 0.0, 1.0, 2.0],
               [0.0, 0.0, 0.0, 1.0]]}

# ソルバーを生成
solver = SoanSolver()

# ソルバーにモデルをセット
solver.set_model(qmat=qmat)

# ソルバーにパラメータをセット
solver.set_param(auto=True, algorithm='ALL', proc= 'GPU')

# 求解
result = solver.solve()

# 結果を表示
print( '* qubits =  ', GetQubits(result))
print( '* energy =  ', GetEnergy(result))
print( '* time   =  ', GetTime(result))

# ソルバーを解放
solver.free()
```

【実行結果】

```
$ python example_2.py
* qubits =  [0, 0, 0, 0]
* energy =  0.0
* time   =  0.034128
```

(c) 係数辞書データによりモデル設定した例

・QUBO モデルの場合

```
# example_c_qubo.py

rom pysoan.util.result import GetQubits,GetEnergy,GetTime
from pysoan.solver.soan_solver import SoanSolver

# # 整数値のキー
# Q = {(0,0):1.0, (1,1):1.0, (2,2):1.0, (3,3):1.0,
#      (0,1):2.0, (0,2):2.0, (0,3):2.0, (1,2):2.0, (1,3):2.0, (2,3):2.0}

# 文字列のキー
Q = {('a','a'):1.0, ('b','b'):1.0, ('c','c'):1.0, ('d','d'):1.0,
     ('a','b'):2.0, ('a','c'):2.0, ('a','d'):2.0, ('b','c'):2.0, ('b','d'):2.0,
     ('c','d'):2.0}

# ソルバーを生成
solver = SoanSolver()

# ソルバーにモデルをセット
solver.set_model(Q=Q)

# ソルバーにパラメータをセット
solver.set_param(auto=True, proc='GPU')
```

```
# 求解
result = solver.solve()

# 結果を表示
print( '* qubits =  ', GetQubits(result))
print( '* energy =  ', GetEnergy(result))
print( '* time   =  ', GetTime(result))

# ソルバーを解放
solver.free()
```

【実行結果】
```
$ python example_c_qubo.py
* qubits =  {'c': 0, 'a': 0, 'b': 0, 'd': 0}
* energy =  0.0
* time   =  0.034651
```

・イジングモデルの場合

```
# example_c_ising.py

from pysoan.util.result import GetQubits,GetEnergy,GetTime
from pysoan.solver.soan_solver import SoanSolver

# # 整数値のキー
# J = {(0,1):2.0, (0,2):2.0, (0,3):2.0, (1,2):2.0, (1,3):2.0, (2,3):2.0}
# h = {0:1.0, 1:1.0, 2:1.0, 3:1.0}

# 文字列のキー
J = {('a','b'):2.0, ('a','c'):2.0, ('a','d'):2.0, ('b','c'):2.0, ('b','d'):2.0,
     ('c','d'):2.0}
h = {'a':1.0, 'b':1.0, 'c':1.0, 'd':1.0}

# ソルバーを生成
solver = SoanSolver()

# ソルバーにモデルをセット
solver.set_model(J=J, h=h)

# ソルバーにパラメータをセット
solver.set_param(auto=True, proc='GPU')

# 求解
result = solver.solve()

# 結果を表示
print( '* qubits =  ', GetQubits(result))
print( '* energy =  ', GetEnergy(result))
print( '* time   =  ', GetTime(result))

# ソルバーを解放
solver.free()
```

【実行結果】
```
$ python example_c_ising.py
* qubits =  {'b': -1, 'a': 1, 'd': -1, 'c': 1}
* energy =  -6.0
* time   =  0.034881
```

(d) PyQUBO データによりモデル設定した例

```
# example_d.py

from pyqubo import Array
from pysoan.util.result import GetQubits,GetEnergy,GetTime
from pysoan.solver.soan_solver import SoanSolver

# # 一次元変数
# x = Array.create('x', shape=(4), vartype='BINARY')
# H = sum(x[i] for i in range(4))**2
# H_compiled = H.compile()
# qubo, offset = H_compiled.to_qubo()

# 二次元変数
x = Array.create('x', shape=((2,2)), vartype='BINARY')
H = sum(sum(x[i][j] for j in range(2)) for i in range(2))**2
H_compiled = H.compile()
qubo, offset = H_compiled.to_qubo()

# ソルバーを生成
solver = SoanSolver()

# ソルバーにモデルをセット
solver.set_model(pyqubo=(qubo, offset))

# ソルバーにパラメータをセット
solver.set_param(auto=True, proc='GPU')

# 求解
result = solver.solve()

# 結果を表示
print( '* qubits =  ', GetQubits(result))
print( '* energy =  ', GetEnergy(result))
print( '* time   =  ', GetTime(result))

# ソルバーを解放
solver.free()
```

【実行結果】

```
$ python example_d.py
* qubits =  [[0, 0], [0, 0]]
* energy =  0.0
* time   =  0.034536
```

※注釈
(*1)「目的関数」という用語について：組合せ最適化問題は、ある制約条件の下で目的関数を最小（または最大）にするための変数の組を求める問題である。従来の数理最適化ソルバーでは、その目的関数と制約条件を別々に数式表現し、その各々を入力することにより解を求めることが行われている。しかし、アニーリング法においては、必ずゼロにならなければならない関数として制約条件を記述しておき、もとの目的関数に足し込むことで新たな目的関数を構成し、それを最小（または最大）にする解探索を行う。本マニュアルでは、制約条件を含めた目的関数を「目的関数」という用語で記載している。

(*2)PyQUBO：（株）リクルートが開発しているオープンソースの Python ライブラリ。目的関数の多項式表現から係数データを取得できる。D-Wave の SDK にも同梱されており、アニーリングの前処理ツールとして広く利用されている。
(参考)https://github.com/recruit-communications/pyqubo

(*3)size の最大値が数万～4 万程度を超える問題を扱いたい場合は、問題を分割して複数の小さい問題を解いて、それらを統合して最終解を得る等の工夫が必要になる。また、係数の値と問題のサイズによっては途中計算がオーバーフローしてエラーが発生する場合がある。そのような場合、係数すべての値を定数倍して小さくするか、あるいは、問題分割する等の工夫が必要になる。

(*4)anneal_time、troter、repetition の最大値は各々10,000,000、1,000、1,000 に設定されている。これらの値を大きく設定すると、最終的に得られる解の精度は良くなるが、その分処理時間が長くなる。解きたい問題に応じて適切に設定する必要がある。まずはデフォルト設定で計算してみて、満足できなかった場合、調整してみるのが良い。

(*5)アニーリング法は厳密解を時間をかけて取得する手法ではなく、確率的に近似解を取得する手法であるため、解きたい問題によっては実行のたびに結果が異なる場合がある。

3　注意事項

・ライセンスについて

本ソフトウェアを使用するにあたり、ライセンス契約（「DNP アニーリング・ソフトウェア」使用許諾契約書）が必要であり、使用者は契約者に限る。

本ソフトウェアを利用する際のその他の注意事項についても、上記ライセンス契約の条項に準ずるものとする。

・各パラメータの範囲について

- set_model メソッドで指定できるモデルのサイズの範囲は 1～40000 である。
- set_param メソッドで指定できるアニーリング時間（anneal_time オプション）の範囲は 1～10000000 である。
- set_param メソッドで指定できるトロッターサイズ（trotter オプション）の範囲は 1～1000 である。
- set_param メソッドで指定できる繰り返し数（repetition オプション）の範囲は 1～1000 である。

以上