汎用リコメンドシステム仕様検討

（株）数理システム　数理計画部

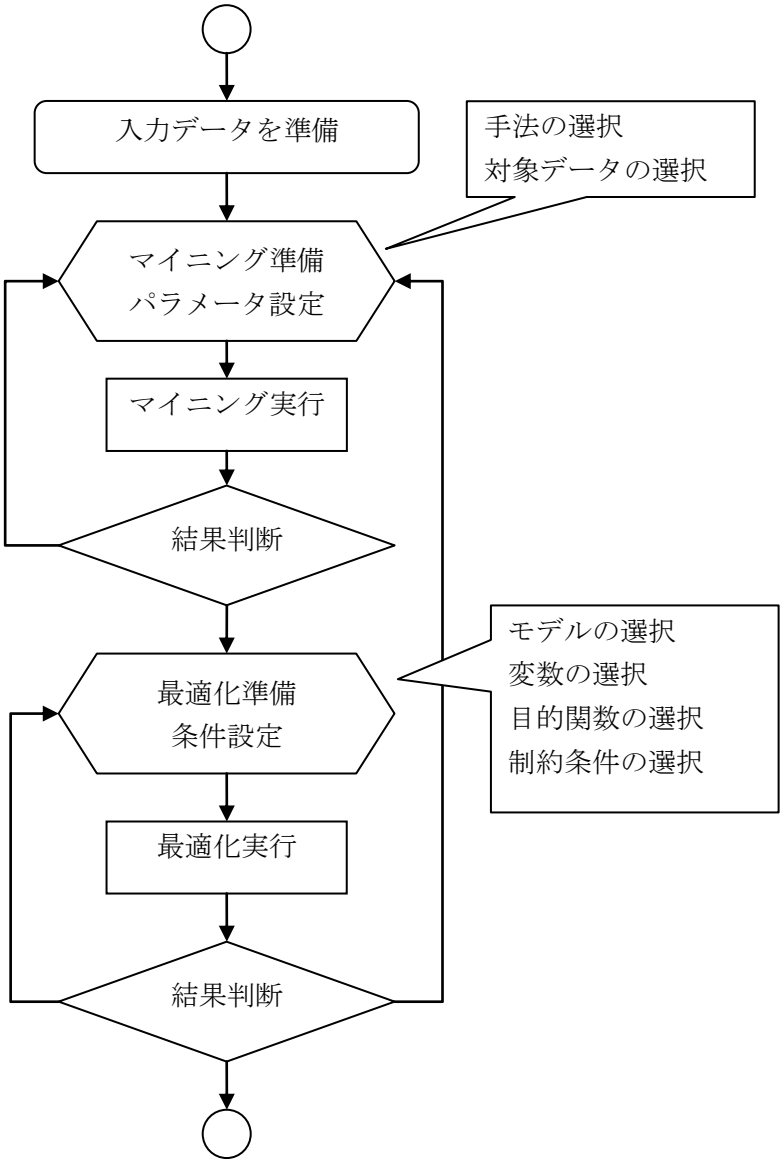
目次

1. システムの概要

システムは「マイニング部分」と「最適化部分」に大別される．マイニング部分は実データを加工して最適化部分の入力データを作成するフェーズである．具体的にどのようなデータを作成する必要があるかは後述する．最適化部分はマイニング部分で作成されたデータをもとに最終結果を吐き出すものである．

1.1 実行の流れ

汎用リコメンドシステムといっても結局は「マイニングして最適化する」という流れを自動化しただけのものである．しかしながらエンドユーザからすると，マイニングも最適化も非常に高度な技術であり，それらを組み合わせたものというのはそれだけで技術的に高度なものであると考えられる．以下エンドユーザからみたシステムの実行フローチャートを示す．



ユーザが行う最大の作業は「入力データの準備」となる．入力データが存在しなければシステム側は何もできない．

マイニング準備ではマイニング手法を選択したり，マイニングの際に用いるパラメータ設定等を行う．また，入力データの中のどのデータを用いるかというのもここで選択すると考えている．

最適化準備では目的関数の選択（利益の最大化・平準化，予算の最小化）や制約条件の選択（予算上限等）を行う．

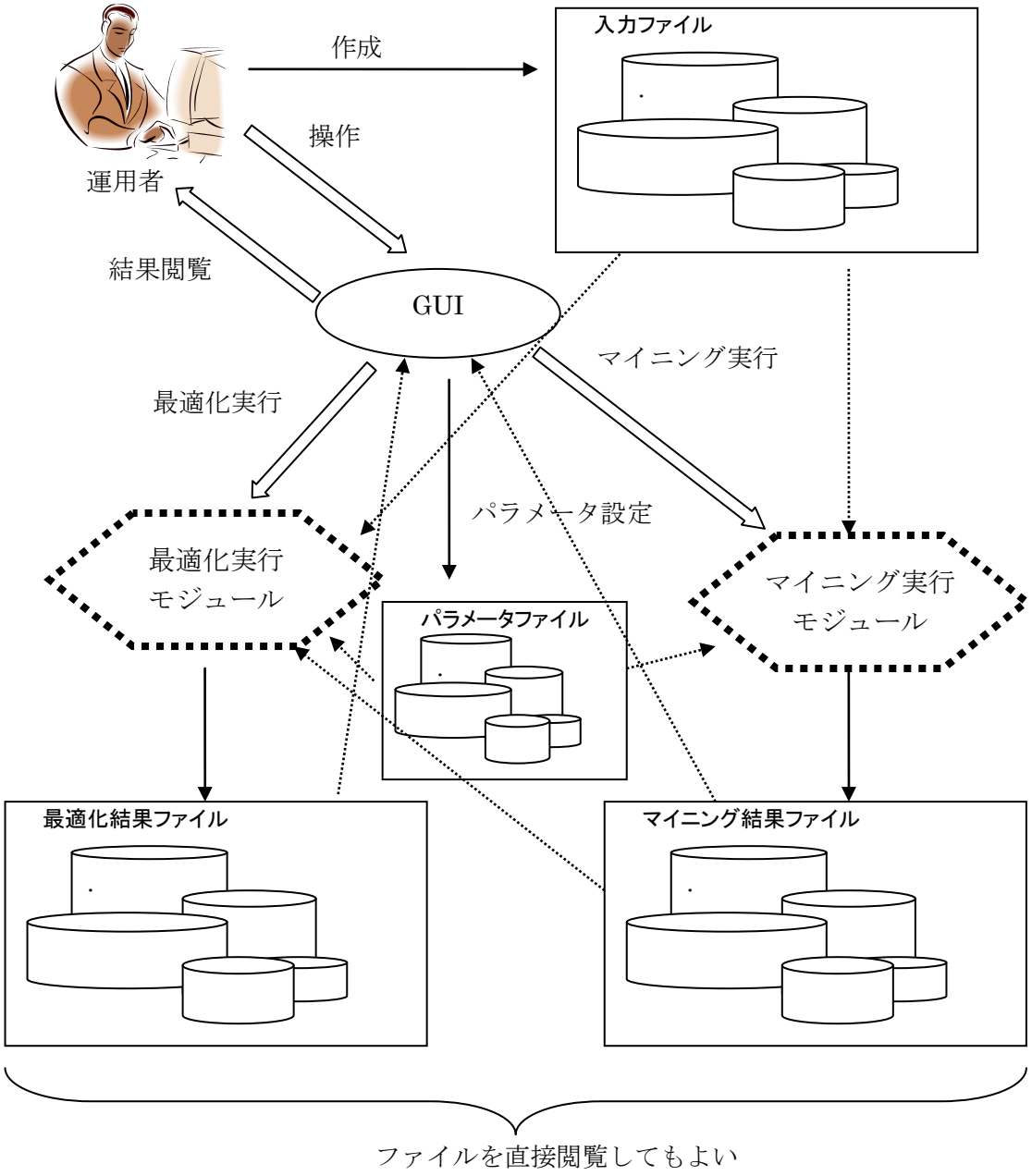
出力された結果を検討して，最適化条件を変更して解きなおすことや，マイニングから解きなおすことが考えられるため，そういった作業を容易に行えるようなインターフェイスが好ましいと考えられる．

1.2 システム構成

これまでの仕事を通して，リコメンドシステムに対してエンドユーザから GUI の要望に関して以下のような意見が出ている．

(1) 広告を配信するのは非常にデリケートな話であるため，人を納得させるために綺麗な GUI があった方がよい．
(2) 扱うデータが非常に大規模ということもあり，GUI で結果を閲覧したりということは特に希望しない．むしろグラフで集計された結果さえ見られれば，詳細はファイル出力でかまわない．
(3) 現在検証用のプログラムがあるため，結果ファイルさえあればあとは不必要である．よって実行やパラメータの設定が容易であれば他は特に重要ではない．

上記要望を加味すると (1) を除いて GUI は特に必要としていない．このリコメンドシステムに関しては GUI よりもおそらく他のアプリケーションに組み込みやすい形であった方がよいと考えられる．例えば，マイニング・最適化部分を切り出して exe 形式にしておき，ファイルをインターフェイスとして Excel で制御するというのが 1 つの手である．まずはエンジン部分に尽力するのが得策であると考えられる．以下，システム構成例をイメージ図で紹介する．



「入力ファイル」というのは，ユーザが用意するものを指す．例えば商品の売り上げ高と実際に打った広告費用が時系列で揃っているもの等である．

2. 最適化部分

リコメンドシステムの最適化部分に関しては大きく二つのモデルがあると考えられる. 本稿の冒頭で述べたメディア列挙型と予算振り分け型である. ここではその二種類について仕様をまとめる.

2.1 メディア列挙型

広告を打つメディアの候補を（種類や時期・予算）全て列挙しておいて，予算の範囲内で実際に列挙されたどのメディアに投資するかを決定する問題. 適用先としては以下のような性質を持った問題が相応しい.

(1) テレビの枠を買うなど，広告の打ち方が非常に分かりやすい.
(2) 打った広告に対して比較的レスポンスがはかりやすい.
(3) Web 等 でよくある話だが，キーワードに格付けするだとか，ランク付けの話.

割り当て問題として定式化することができ, 複数のナップザックがあるナップザック型で表現することができる．以下定式化の大枠をまとめる.

＜集合＞

I ：メディアの集合．添え字は i.

J ：商品の集合．場合によっては店舗であるとか,（広告代理店から見た）広告主と設定することもできる．添え字は j.

＜変数＞

x_{ij} ：メディア i に商品 j の広告を打つならば 1 ，そうでないならば 0.

＜定数＞

$price_i$ ：メディア i に広告を掲載する際の費用（場合によっては添え字 i, j）.

$gain_{ij}$ ：メディア i に商品 j の広告を打った際の効果.

$yosan_j$ ：商品 j に使うことのできる予算.

$total\_yosan$ ：トータルの予算.

＜制約条件＞

○ 各メディアには商品を高々 1 つ割り当てることができる.

$$\sum_j x_{ij} \leq 1$$

○ 予算制約

$$\sum_i price_i x_{ij} \le yosan_j, \sum_{i,j} price_i x_{ij} \le total\_yosan$$

＜目的関数＞

○ 効果の最大化

$$(\text{maximize}) \sum_{i,j} gain_{ij} x_{ij}$$

上記は非常にシンプルな定式化の例である.以下汎用化に向けた問題設定の応用を考えてみる.

2.1.1 予算に関する応用

予算制約に関しての応用の幅を検討する．シンプルな定式化では上限のみの設定であったが,広告代理店側からすると「予算は全て消化したい」と考える場合もある．それは以下のような定式化で表現することができる.

MIP の場合の定式化

＜変数＞

s_j^+, s_j^-, ss^+, ss^- ；非負のスラック変数.

＜制約式＞

$$\sum_i price_i x_{ij} = yosan_j + s_j^+ - s_j^-$$

$$\sum_{i,j} price_i x_{ij} = total\_yosan + ss^+ - ss^-$$

＜目的関数＞

$$(\text{maximize}) \sum_{i,j} gain_{ij} x_{ij} - \alpha \sum_j \left(s_j^+ + s_j^- \right) - \beta \left(ss^+ + ss^- \right)$$

※α, β は定数

WCSP の場合の定式化

＜SoftConstraint＞

$$\sum_i price_i x_{ij} = yosan_j \quad , \quad \sum_{i,j} price_i x_{ij} = total\_yosan$$

また，商品ごとの予算の消化の平準化に関し，WCSP では 2 乗のペナルティを考慮することもできる．これによりある商品に予算制約の違反を押しつけることがなくなる.

2.1.2 利得に関する応用

広告代理店から眺めた場合，全体の利得を最大化することはさほど利益に繋がらない．各商品（広告主と考えるとイメージしやすい）の満足度を満たすということが重要になるという考え方もある．まず目標値を設定する場合の定式化方法について説明する．

MIP の場合の定式化

＜定数＞

$gain\_target_j$ ：利得の目標値．

＜変数＞

s_j^+, s_j^- ；非負のスラック変数．

＜制約式＞

$$\sum_i gain_i x_{ij} = gain\_target_j + s_j^+ - s_j^-$$

＜目的関数＞

$$(\text{maximize}) \sum_{i,j} gain_{ij} x_{ij} - \alpha \sum_j \left(s_j^+ + s_j^- \right)$$

※α は定数

WCSP の場合の定式化

＜SoftConstraint＞

$$\sum_i gain_{ij} x_{ij} = gain\_target_j$$

こうした場合入力として $gain\_target$ を設定する必要がある．広告主（商品）がメディア候補集合をある程度知っていた場合，広告主（商品）から見た $gain\_target$ として妥当と考えられるのは，他の広告主（商品）と競合せずに予算の範囲内で自由にメディアに割り当てできる場合の結果である．よって以下のように $gain\_target$ を算出することもできる．以下の問題は j 毎に切り分けて解くことができる．

＜変数＞

y_i^j ：商品 j をメディア i に割り当てるならば 1 そうでないならば 0 ．

＜制約条件＞

$$\sum_i price_i y_i^j \le yosan^j$$

＜目的関数＞

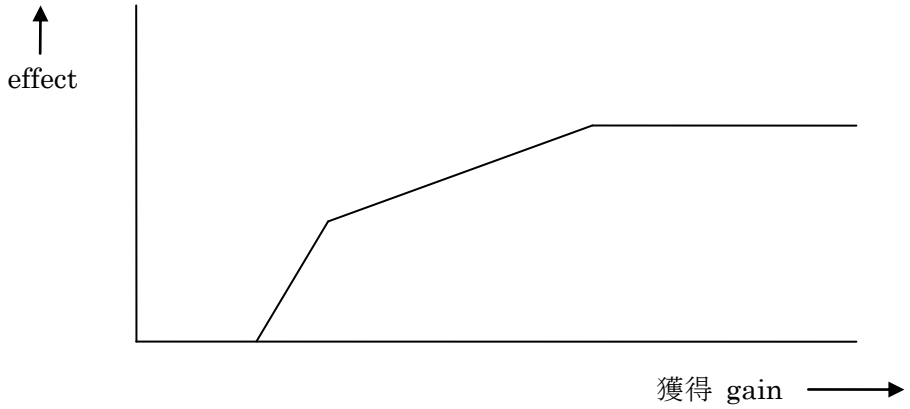
$$(\text{maximize}) \sum_i gain_i^j y_i^j$$

上記問題の目的関数の値が $gain\_target$ となる．

2.1.3 効果の“saturate”の表現

広告効果として定数 *gain* というものを考慮してきた．この *gain* については獲得すればするほど効果が増えるという単純な加算で考慮してきたが実際には *gain* 獲得量と実際の効果は比例しないであろう．単純な話，来客数と売り上げを考えるとある一定の来客数までは売り上げと直結すると考えられるが，店が捌ききれない程の来客があった場合にはそうはいかないだろう．

以下のような折れ線で獲得 *gain* と実際の効果 *effect* で関連付けされているとする．



離散問題であることを考えると線形でモデル化することは必須である．これを MIP で表現することは比較的容易ではある．連続変数と 0 – 1 整数変数を幾つか導入することによって定式化することができる．

しかしながらこの問題は大規模になることが予想できるため WCSP で求解することが考えられるだろう．この問題を 0-1 整数変数のみで定式化すると， 2 次の項が出現してしまう．WCSP は 2 次の項が出てきても高速に計算を行うことは可能であるが，変数の数が増えてしまうことは避けられない．よって，以下の手法をとることによって高速な処理を行うことが期待される．

> WCSP の新関数で折れ線応答というのを作成すればよいのでは！
> これはもしかしたらリコメンドシステム云々ではなくもっと汎用的に役に立つかもしれない．

このアイディアについては本稿の筋とは少しずれてしまうので本稿末の付録にて掲載する．

2.1.4 予算振り分け型をメディア列挙型で表現

順番が前後するが，次章で紹介する予算振り分け型をメディア列挙型で表現することも可能である．メディアを列挙する際に，予算も一区切りでメディアと考えてしまう方法である．例えば「新聞に 100 万」をメディア 1 とし，「新聞に 200 万」をメディア 2 ... として考え，それぞれの *gain* を準備しておけばよい．その場合，以下の制約式を追加することで問題の表現をすることができる．

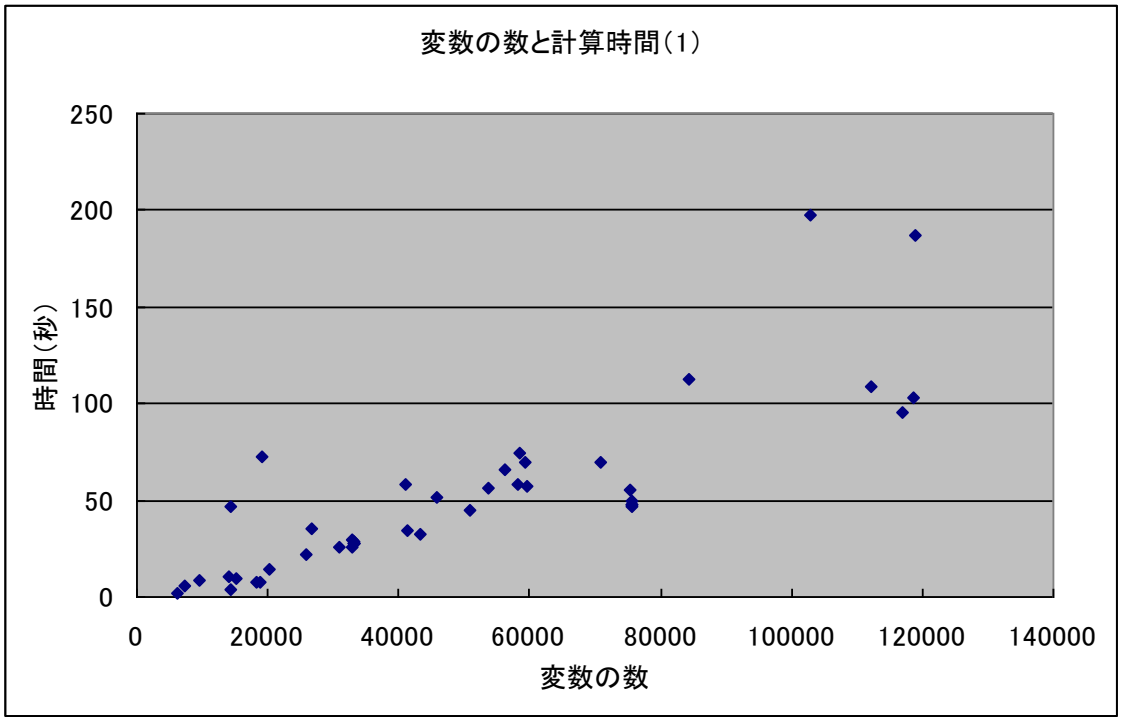
＜制約式＞

$$\sum_{j,i\subset 新聞} x_{ij} \leq 1$$

2.1.5 メディア列挙型まとめ

メディア列挙型で重要となる入力は *gain* である．このテーブルが適切に作成できるならばメディア列挙型というのは有用な手法であろう．テレビを考えたときに「チャンネルは？」「放送時間帯は？」とうの疑問が出るかもしれないが，テレビの各放送枠を１つのメディアとして考慮するために，そのような情報は全て *gain* が持っていると考えることができる．

問題規模とパフォーマンスについては 2007 年度の技術レポート「数理計画法パッケージ NUOPT を用いた割り当て問題の解法に関する実践的解説（佐藤　誠)」を参照されたい．以下は 2007 年度の技術レポートに掲載した割り当て問題の計算パフォーマンスのグラフである．



この問題は施設配置問題を WCSP を用いて解いたものであり，3600 秒計算をまわして最終更新時間を取ったものである．

2.2 予算振り分け型

どの時期にどのメディアにどういった広告をいくら打つかという意思決定を行う問題である．以下の図の空白部分のマスが変数となり，配分予算を決定する．

	数理新聞広告			数理テレビ			佐藤テレビ		
	商品A	商品B	商品C	商品A	商品B	商品C	商品A	商品B	商品C
2009年2月									
2009年3月									
2009年4月									
2009年5月									
2009年6月									
2009年7月									
2009年8月									
2009年9月									
2009年10月									
2009年11月									
2009年12月									
2010年1月									
2010年2月									
2010年3月									
2010年4月									
2010年5月									
2010年6月									
2010年7月									
2010年8月									
2010年9月									
2010年10月									
2010年11月									
2010年12月									
2011年1月									
2011年2月									
2011年3月									

マイニング部分では過去の広告実績から広告費用と効果を表す関係を導出しておく．上図の「数理新聞広告」「数理テレビ」「佐藤テレビ」はメディアを表しており，「商品 A 」「商品 B 」「商品 C」は広告の種類を表している．もっと細かい計画を建てたい際にはメディアを「数理テレビ・深夜枠」のように準備することで対応できる．以下定式化の大枠をまとめる．

＜集合＞

I ：メディアの集合．添え字は i ．

J ：商品の集合．添え字は j ．広告の種類と考えて i, j のペア集合を用いてもよい．

T ：時間の集合．添え字は t ．

＜変数＞

x_{tij} ：メディア i に商品 j の広告を時刻 t のときに打つ量．

＜定数＞

$gain_{tij}$ ：メディア i に商品 j の広告を時刻 t に打った際の単位量あたりの効果．

$yosan^{u}_{tij}$ ， $yosan^{l}_{tij}$ ：各予算の上下限．

$total\_yosan$ ：合計金額の予算上限値．

＜制約条件＞

○ 予算制約.

$$yosan_{tij}^{l} \le x_{tij} \le yosan_{tij}^{u}$$
$$\sum_{t,i,j} x_{tij} \le total\_yosan$$

＜目的関数＞

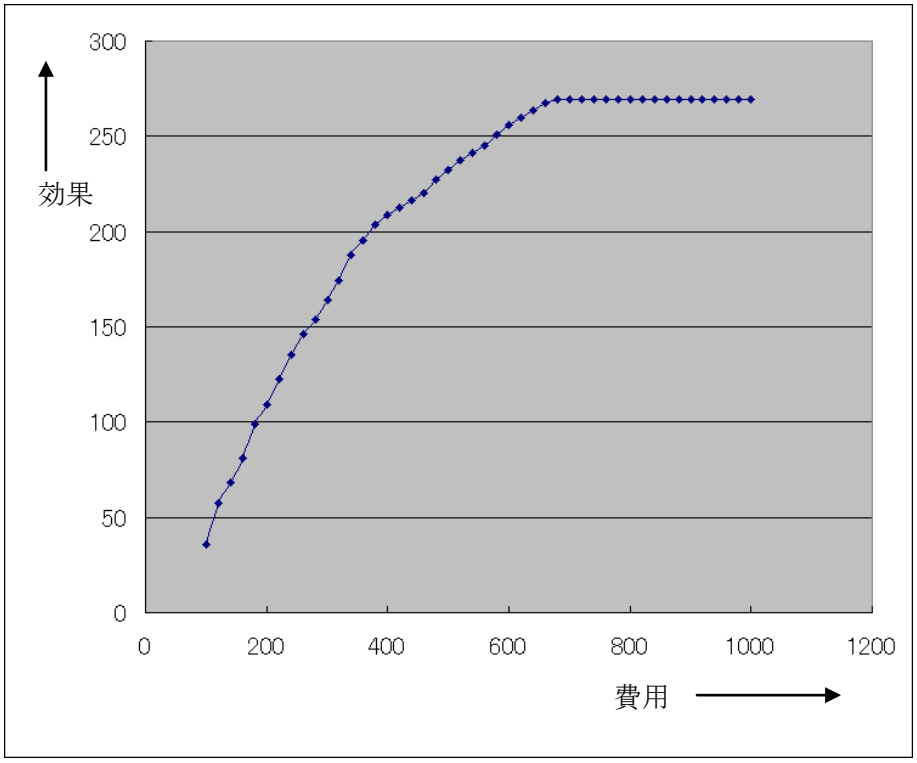
○ 効果の最大化.

$$(\text{maximize}) \sum_{t,i,j} gain_{tij} x_{tij}$$

上記は非常にシンプルな定式化の例である. 以下汎用化に向けた問題設定の応用を考えてみる.

2.2.1 費用対効果曲線の描画

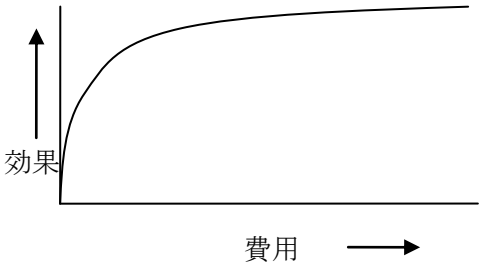
これはモデルの話ではなく運用の話になってしまうが, 予算上限の値を少しずつ増やしていき, 最適化を行い，それに伴う効果との値を見るものである．これは全体の効果を眺めてもよいし, 各メディアについて眺めるのも面白いと思う．以下が費用対効果曲線のイメージである.



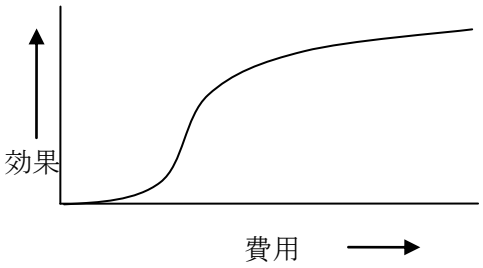
これはメディア列挙型タイプでも同様の解析をすることができる.

2.2.2 効果の“saturate”の表現

メディア列挙型と同様，効果の saturate というのが考えられる．メディア列挙型では，ある広告にある商品を割り当てた際の効果は決定しており，ある商品の累積効果に対しての saturate を考えた．ここではある広告にあてる量（金額と考えるとイメージしやすい）そのものを決定するため，定数である *gain* に saturate が存在するという考えをとる（もちろん商品や広告ごとに集計して考慮してもかまわない）．始めに示した定式化では金額と効果は比例関係であり，*gain* はその比例定数として考えていた．以下のような応答をマイニングの結果から得たとする．



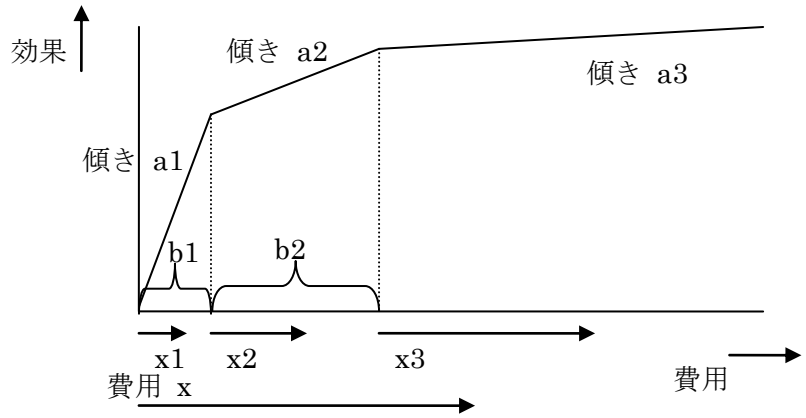
saturate グラフ１



saturate グラフ２

これ自体が *gain* となるのだが，どう考えても非線形応答となる．ある関数を想定しおいて，この関数のまま取り扱うことも可能であるが，その場合は大規模非線形問題となってしまうので注意が必要である．「 saturate グラフ１」は上に凸であるため取り扱いはある程度容易（それでも非線形問題にはなる）であるが，「 saturate グラフ２」に関しては凸性がないため非常に取り扱いが困難である． SimpleExternalFunc を用いなければいけない程難しい関数を用いることになってしまうため，大規模問題であることを考えると現実的に取り扱うのは絶望的と考えた方がよいだろう．

ここでは，「 saturate グラフ１」のように凸である関数系を仮定して，それを折れ線関数に近似して取り扱うことを考えてみる．この場合，凸性が保証されているので離散変数を追加することなく連続変数の追加のみで実現が可能である．以下のような折れ線近似がされたとして説明をする．



上図では以下のように定式化することによって折れ線を表現することができる.

$$x = x1 + x2 + x3$$

$$0 \le x1 \le b1 \quad , \quad 0 \le x2 \le b2 \quad , \quad 0 \le x3$$

$$\text{効果} = a1 \cdot x1 + a2 \cdot x2 + a3 \cdot x3$$

この定式化でうまくいくためには凸性, すなわち微係数の減少 $a1 \ge a2 \ge a3$ が保証されている必要がある. また, 目的関数を見て効果が大きいほどうれしい, という条件が必要である. 今回のシステムでは効果を平準化するということも考慮する可能性があり, その場合は意図通りに作動しないため, 充分に注意が必要である. 複雑な関数を準備して局所的最適解に落ち込むことを考えれば, 連続変数のまま線形で扱えるこちらの方が有用ではないかと思う.

2.2.3 細かな制御をする方法

あるメディアにある時期に商品の広告を打つときに, 商品 A を 20 商品 B を 15 というような決定が考えられない（どちらかに打ったらどちらかには打たない）場合は,「ある時期にあるメディアに打つ商品は１つだけ」という制約が必要になる. この場合どうしても 0-1 整数変数が必要になってしまうため扱いには充分に注意が必要である. 特に効果が saturate することを表現するために非線形問題として取り扱う場合には整数変数を入れることができないので注意が必要である.

2.2.4 広告効果の継続の表現

一度広告を打ったあとに効果が継続するということを考慮することも可能である. 一時間区分での忘却率をγとすると以下のように表現することができる.

$effect_t$ ：各時刻での広告効果.

$gain_t$ ：各時刻で累積効果の値.

$$gain_t = \sum_{t'}^{t} \gamma^{t-t'} effect_{t'}$$

上記の累積効果と 5.2.2 の saturate 効果を組み合わせて実際の売り上げ等を表現することができる．この問題を組み込むと，時間の区分が増加した際に，制約式の展開が非常に時間がかかってしまうので注意されたい．

2.2.5 商品・店舗ごとの売り上げの考慮

商品・店舗ごとの売り上げの考慮をしたいということが考えられる．商品の場合は今回取り扱っているモデルの例がそのままなのだが，店舗の場合は少し考える必要がある．

１つの方法としては $gain$ の添え字に店舗集合を付加して店舗ごとの効果として集計すればよい．

いずれにせよ，商品・店舗ごとの売り上げが得られたと仮定して考える（これ以降は店舗としてのみ表記する）．店舗は規模がまちまちであり純粋に値を平準化するのは非現実的であり，店舗毎の目標値を定めてそれの満たし具合を考慮するのが妥当だと思われる．以下定式化をまとめる．

＜集合＞

K ：店舗集合．添え字は k

＜変数＞

s_k ：スラック変数

＜式（Expression）＞

$shop\_gain_k$ ：店舗毎の利得．

＜定数＞

$shop\_target_k$ ：店舗毎の利得の目標値

＜制約式＞

$$shop\_gain_k - shop\_target \geq -s_k$$

＜目的関数＞

$$(\text{minimize}) \sum_k s_k$$

上記の制約式ではどこかの店舗にしわ寄せ現象が起きる可能性がある．これを解決するためには目的関数で s の二乗和を取ればよいのだが問題の規模等考えるとできれば線形で表現したいところである．

また，この制約式では規模の大きい店舗から優先的に考慮される（目的関数に与える影響が大きいため）ことが考えられる．以下のように目標値に対する割合でスラック変数を考慮することもできる．

＜制約式＞

$$shop\_gain_k / shop\_target_k - 1 \geq -s_k$$

2.2.6 予算振り分け型まとめ

予算振り分け型は当然予算の分配が目的であるため，連続変数で考えるのが合点がいく．よって整数変数が入るような定式化はできれば避けたいところである．ここでのまとめとして，今まで紹介した幾つかの方法を加味したデラックスな定式化をまとめる．

＜集合＞

I ：メディアの集合．添え字は i．

J ：商品の集合．添え字は j．広告の種類と考えて i, j のペア集合を用いてもよい．

K ：店舗集合．添え字は k．

T ：時間の集合．添え字は t．

F ：費用に対する効果の折れ線の折れる数．添え字は f．

＜変数＞

x_{tij} ：メディア i に商品 j の広告を時刻 t のときに打つ量（式として表現可能）．

y_{tijf} ：折れ線を考慮して効果を表現するための中間変数．

s_{tk} ：店舗の売り上げ平準化のためのスラック変数．

＜式（Expression）＞

$effect_{tij}$ ：各時刻のメディア i の広告効果による商品 j の累積広告効果．

$gain_{tij}$ ：各時刻のメディア i の広告効果による商品 j の広告効果．

$shop\_gain_{tk}$ ：店舗毎の利得．

＜定数＞

a_{tijf} ：メディア i に商品 j の広告を時刻 t に打った際の折れ線の傾き．

b_{tijf} ：折れ線の幅．正確には添え字 f の最後は不要．

$yosan^u_{tij}$，$yosan^l_{tij}$ ：各予算の上下限．

$total\_yosan$ ：合計金額の予算上限値．

$shop\_sence_{ijk}$ ：メディア i に打った商品 j の累積広告効果に対する店舗の感度．

$shop\_target_{tk}$ ：店舗毎の利得の目標値．

M ：大きな値

γ ：広告効果の１時間区分あたりに残る量．

＜制約条件及び式の定義＞

○ 予算制約

$$yosan_{tij}^{l} \le x_{tij} \le yosan_{tij}^{u}$$

$$\sum_{t,i,j} x_{tij} \le total\_yosan$$

○ 折れ線の表現

$$x_{tij} = \sum_{f} y_{tijf}$$

$$0 \le y_{tijf} \le b_{tijf}$$

$$gain_{tij} = \sum_{f} a_{tijf} y_{tijf}$$

○ 累積効果の表現

$$effect_{tij} = \sum_{t'}^{t} \gamma^{t-t'} gain_{tij}$$

○ 店舗の考慮

$$shop\_gain_{tk} = \sum_{ij} shop\_sence_{ijk} effect_{tij}$$

$$shop\_gain_{tk} / shop\_target_{tk} - 1 \ge -s_{tk}$$

＜目的関数＞

○ 効果の最大化及び店舗毎の目標の達成

$$(\text{maximize}) \sum_{t,i,j} gain_{tij} - M \sum_{tk} s_{tk}$$

ここで紹介した定式化の例を NUOPT 付属のモデリング言語 SIMPLE で記述したものを付録に掲載する．モデリングや計算パフォーマンスについては付録を参照されたい．

2.3 モデル化の際の定式化の工夫

本稿で取り扱うような問題は，業務で使用する際には大規模問題となることが予想される．ここでは紹介した二つのモデルに共通した定式化の際の工夫を紹介する．

2.3.1 変数の削除

「ある商品は絶対にこの広告に割り当てない」「一定以上効果が小さい組み合わせには割り当てない」「もう既に割当先や，割り当てる料金が決まっているものがある」などのルールがある場合は事前に入力データからその部分をカットしておくことによって変数の数を減らすことが

できる．NUOPT 付属のモデリング言語 SIMPLE では superset を用いることによって記述することができる．その結果，計算時間の短縮，制約式展開時間の短縮，メモリーオーバーをある程度避けるといった効果が期待できる．

2.3.2 変数の固定

2.3.1 に該当する場合に適用することができる事柄なので，変数を削除することで対応ができる話ではある．しかしながら，割り当てる広告費用が一部決定している場合に変数を削除する場合には予算の上下限の設定しなおしや，目的関数の再集計が必要となる．これらが面倒な場合には変数を固定してしまうことが考えられる．SIMPLE では fixVariable() を用いることによって実現可能である．これによって，計算時間の短縮，メモリーオーバーをある程度避けるといった効果が期待できる．

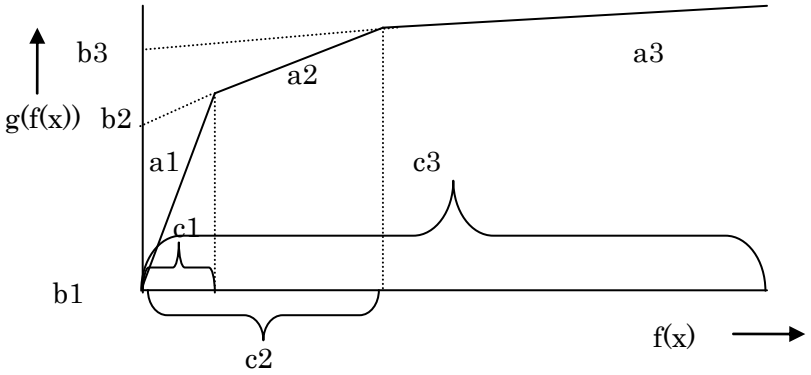
2.3.3 問題を切り分ける

一部の商品は絶対「テレビ」にしか割り当てない，しかもテレビ割り当てる商品は必ずテレビ以外には割り当てられない，といったように一部分で閉じた問題を作れる場合に適用することができる．最適化モデルの外側でパラメータを整理・分割し，繰り返しとくことをすればよい．この方法は大規模問題を幾つかに分割できるため本来取り扱うことのできない問題も取り扱えるようになるほど強力なものである．

— 以上 —

付録 1 WCSP の新関数 pwl() 仕様検討

新関数候補 WCSP 特有の関数，折れ線関数の応答についての仕様をここでは検討する．これは **2.1.3 効果の“saturate”の表現** の部分で述べているものである．以下折れ線関数のイメージ図を示す.



上記の f(x) は SIMPLE の Expression で定義されているものであり，g は上図の折れ線応答を表している．a は各区間の傾き，b は切片，c は区間長を表す．この g を表現するために現状では変数を増加することによって対応することが可能である．以下，変数増加による表現方法について述べる.

＜集合＞

I ：折れ線の折れる回数の集合．添え字は i .

＜変数＞

y_i ： $f(x)$ の値がが折れ線区間にいるかどうかのインディケータ変数.

＜制約式＞

$$f(x) \le c_i y_i \quad , \quad f(x) \ge c_i y_{i+1}, i \ne I.last()$$

$$\sum_i y_i = 1$$

＜式（Expression）＞

$$g = \sum_i y_i \left(a_i f(x) + b_i \right)$$

このように定式化すると，折れ線の折れ回数分だけ 0-1 整数変数が必要になり，制約式も幾つか追加される．g の表現に二次式を考慮しているが，これは WCSP の仕様上，速度低下の直接の原因にはならない．この f(x) が何本も存在し，全てが応答 g に従うようなときには，それぞれの f(x) にインディケータ変数を準備する必要があるので，変数の増加という意味でも求解パフォーマンスが懸念される.

新関数を導入する最大のメリットは「変数の数，制約式の数を増加させることなく表現

可能であり，探索空間や近傍をみだりに増加させることはない」といえる．

以下，新関数の仕様を検討する．対象となるのは f(x) が x の線形の関数で表現されているときである．

＜インターフェイス＞

```
pwl(IntegerVariable x,Element i,Parameter d,Element j,
            Parameter a,Parameter c,Element k)
```

上記仕様は x[i] という変数があり，f[j] = sum(d[i,j]*x[i],i) と対応付けされていることを想定している．また傾きは a[k] で，折れ線区間は c[k] で与えられている．切片 b[k] は傾きと折れ線区間から導出できる．

＜アルゴリズム＞

大雑把に言えば f[j] = sum(d[i,j]x[i],i) の値がどの区間に入っているかを高速で見つけることが重要となる．以下アルゴリズムを示す．簡単のため f[j] を１つしかないものであり，変数の変化は 0 -> 1 であるとする．変数の変化が逆の場合は符号を逆にして考えればよい．

(0) f[j] g[j]の初期値，現在の f[j] の存在する区間 k をそれぞれ記憶し高速アクセス可能なデータ構造を作成．

(1) x[i] が変化した．対応する d[i,j] を見に行き d[i,j]=0 ならば何もしない．そうでないならば，f[j]+d[i,j] を調べ，c の境界を越えるかチェック．超えるならば (2) 超えなければ (3) へ．

(2) f[j] g[j] をそれぞれ普通に導出して終わり．

(3) f[j] を f[j]+d[i,j] に更新．g[j] を g[j]+d[i,j]*a[k] に更新して終わり．

工夫の余地としては (1) の 「c の境界を越えるかチェック」の部分で，各 f[j] から現在の位置から自分の両サイドの壁の値をアクセスできるようにしておくことが考えられる．その場合 (2) の部分で壁の値（ここでは片方でいい）の更新も必要になる．いずれにせよ，それほど難しくない工夫で普通に定式化するより高速に動作するものを，作りこむことができそうである．

付録 2 予算振り分け型　モデル記述と計算パフォーマンス

ここでは本稿「5.2.6 予算振り分け型まとめ」で紹介した定式化について，具体的な SIMPLE の記述と計算パフォーマンスについて紹介する．ここで紹介する SIMPLE の記述はこのレポート用に作成したものであるため，流用することについてはかまわない．但し，何か不具合等が起きたときや全ての責務については自己責任とさせて頂く．また実際の業務で転用する場合には一言お声をかけて頂きたい．

ここで紹介する SIMPLE のモデルは乱数を用いたデータの作成機能も実装してある．この手のモデルはモデル構築の手間よりもデータ作成の方が時間がかかることが考えられ，モデルのみを紹介しても手元で再現するのが大変なので，そのように準備をした．与えるパラメータは「メディア」「商品」「店舗」「時間」「折れ線の折れ」の個数のみであとはデータを自動生成することができる．自身でデータを準備して実行したい場合にはモデルを適宜書き換えて行って頂きたい．

```
/*************************************************/
// 2008 年度　技術レポート付録
// 予算振り分け型
// 作者：佐藤誠
/*************************************************/

////////////////////////////////
// 集合の宣言
////////////////////////////////
Set I;// メディアの集合
Set J;// 商品の集合
Set K;// 店舗の集合
OrderedSet T;// 時間の集合
OrderedSet F;// 費用に対する効果の折れ線集合

////////////////////////////////
// 添え字の宣言
////////////////////////////////
Element i(set = I);
Element j(set = J);
Element k(set = K);
Element t(set = T);
Element t_(set = T);
Element f(set = F);

////////////////////////////////
// 変数の宣言
////////////////////////////////
// Variable x(name = "広告量",index = (t,i,j));
// x は本モデルの中心となる変数であるが，y を用いて
// 表現できるためここでは Expression として取り扱う．
Variable y(name = "折れ線変数",index = (t,i,j,f));
Variable s(name = "スラック変数",index = k);

////////////////////////////////
// 定数の宣言
////////////////////////////////
Parameter a(name = "折れ線傾き",index = (t,i,j,f));
Parameter b(name = "折れ線幅",index = (t,i,j,f));
Parameter yosan_u(name = "予算上限",index = (t,i,j));
Parameter yosan_l(name = "予算下限",index = (t,i,j));
Parameter total_yosan(name = "予算合計");
Parameter shop_sence(name = "広告感度",index = (i,j,k));
Parameter shop_target(name= "店舗毎利得目標値",index = (t,k));
Parameter M(name = "大きい数");
Parameter r(name = "広告減衰値");
```

```
M = 1.0e+5;
r = 0.1;

// 以下はモデル内でデータ作成するためのもの
void makeParameter(Parameter nI,Parameter nJ,Parameter nK,Parameter nT,
                           Parameter nF,Parameter a,Parameter b,
                           Parameter yosan_u,Parameter yosan_l,
                           Parameter total_yosan,Parameter shop_sence,
                           Parameter shop_target);

// 以下の Parameter は外部から入力
Parameter nI;// メディアの数
Parameter nJ;// 商品の数
Parameter nK;// 店舗の数
Parameter nT;// 時間の数
Parameter nF;// 折れ線の折れ回数

makeParameter(nI,nJ,nK,nT,nF,a,b,yosan_u,yosan_l,total_yosan,shop_sence,shop_target);

////////////////////////////////
// 式の宣言
////////////////////////////////
Expression x(name = "広告量",index = (t,i,j));
Expression effect(name = "累積広告効果",index = (t,i,j));
Expression gain(name = "広告効果",index = (t,i,j));
Expression shop_gain(name = "店舗毎利得",index = (t,k));

////////////////////////////////
// 式の定義
////////////////////////////////
// 折れ線変数 y を用いて x を表現
x[t,i,j] = sum(y[t,i,j,f],f);

// 各時刻における効果を折れ線から算出
gain[t,i,j] = sum(a[t,i,j,f]*y[t,i,j,f],f);

// 累積広告量の算出
for(t_ = T.first();t_ < T;t_ = T.next(t_)){
  effect[t_,i,j] = sum(pow(r,t_-t)*gain[t,i,j],(t,t <= t_));
}

// 各店舗が得られる利得の算出
shop_gain[t,k] = sum(shop_sence[i,j,k]*effect[t,i,j],(i,j));

////////////////////////////////
// 制約条件
////////////////////////////////
// 非負制約
y[t,i,j,f] >= 0;
s[k] >= 0;

// 予算制約 1
yosan_l[t,i,j] <= x[t,i,j] <= yosan_u[t,i,j];

// 予算制約 2
sum(x[t,i,j],(t,i,j)) <= total_yosan;

// 折れ線の表現
0 <= y[t,i,j,f] <= b[t,i,j,f];

// 各店舗が得る利得の制約
sum(shop_gain[t,k] - shop_target[t,k],t) >= -s[k];

////////////////////////////////
// 目的関数
////////////////////////////////
Objective obj(name = "目的関数",type = maximize);
obj = sum(gain[t,i,j],(t,i,j)) - M*sum(s[k],k);

solve();
```

ここまでで最適化部分は完結している．続きはデータ自動作成部分となる．

```
%%%%% #include<stdio.h> %%%%%
%%%%% #include<simple.h> %%%%%
int n(int m){
  return rand()%m;
}
// 入力ファイル生成関数
// nI : メディアの数
// nJ : 商品の数
// nK : 店舗の数
// nT : 時間の数
// nF : 折れ線の折れ数
void makeParameter(Parameter nI,Parameter nJ,Parameter nK,Parameter nT,
                           Parameter nF,Parameter a,Parameter b,
                           Parameter yosan_u,Parameter yosan_l,
                           Parameter total_yosan,Parameter shop_sence,
                           Parameter shop_target){
  Set I;
  Set J;
  Set K;
  OrderedSet T;
  OrderedSet F;
  Element i(set = I);
  Element j(set = J);
  Element k(set = K);
  Element t(set = T);
  Element f(set = F);
  Parameter it;
  for(it = 1;it <= nI; it = it + 1){
    I.add((Element)it.val.asDouble());
  }
  for(it = 1;it <= nJ; it = it + 1){
    J.add((Element)it.val.asDouble());
  }
  for(it = 1;it <= nK; it = it + 1){
    K.add((Element)it.val.asDouble());
  }
  for(it = 1;it <= nT; it = it + 1){
    T.add((Element)it.val.asDouble());
  }
  for(it = 1;it <= nF; it = it + 1){
    F.add((Element)it.val.asDouble());
  }
  // 傾きは適当に saturate するように
  Parameter it_i;
  Parameter it_j;
  Parameter it_k;
  Parameter rand_a(index = (i,j));
  Parameter rand_aa(index = (i,j));

  for(it_i = 1;it_i <= nI; it_i = it_i + 1){
    for(it_j = 1;it_j <= nJ; it_j = it_j + 1){
      rand_a[it_i,it_j] = n(10000)/1000.0;
      rand_aa[it_i,it_j] = n(8000)/10000.0+0.2;
    }
  }
  //  rand_aa.val.print();
  a[t,i,j,f] = (1 - f*rand_aa[i,j]/(nF + 1))*rand_a[i,j];
  b[t,i,j,f] = 100;
  yosan_u[t,i,j] = 100*nF;
  yosan_l[t,i,j] = 0;
  total_yosan = sum(yosan_u[t,i,j],(t,i,j))*0.6;
  Parameter rand_ss(index = (i,j,k));
  for(it_i = 1;it_i <= nI; it_i = it_i + 1){
    for(it_j = 1;it_j <= nJ; it_j = it_j + 1){
      for(it = 1;it <= nK; it = it + 1){
            rand_ss[it_i,it_j,it_k] = n(8000)/10000.0+0.2;
      }
    }
  }
  shop_sence[i,j,k] = rand_ss[i,j,k];
  shop_target[t,k] = sum(shop_sence[i,j,k]*5,(i,j));
}
```

以上でモデル部分の紹介は終了である.

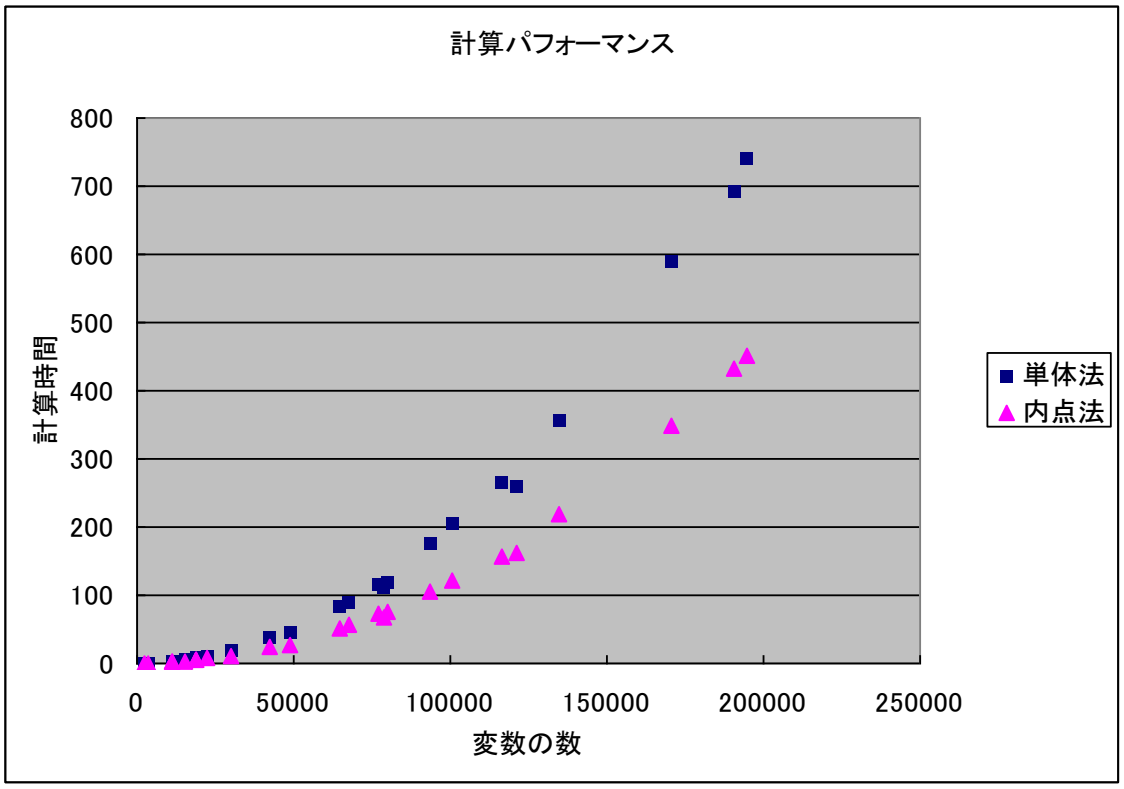
次に変数の数と計算時間の関係についてまとめる．上記モデルは線形計画問題なので NUOPT を用いて解く際は単体法か内点法を選択することができる．ここでは，両者で計算を行い，その差も検証の対象とした．以下に結果をまとめる．

変数の数と計算時間結果

変数の数	単体法	内点法	内点法／単体法
11256	2.78	1.89	0.679856115
11256	2.48	1.72	0.693548387
18756	7.3	4.88	0.668493151
22506	10.59	6.91	0.652502361
18756	7.44	4.8	0.64516129
30011	17.69	11.5	0.650084794
18752	7.33	4.84	0.660300136
3756	0.3	0.28	0.933333333
2256	0.08	0.09	1.125
15011	4.7	3.24	0.689361702
42129	38.72	23.17	0.59839876
48764	45.47	26.98	0.593358258
100815	206.63	122.16	0.591201665
64609	84.92	51.08	0.601507301
93766	175.16	106.22	0.60641699
116651	265.53	157.13	0.591759876
170534	589.38	349.38	0.592792426
67621	88.67	55.72	0.628397429
78776	111.49	67.7	0.607229348
80006	119.73	74.98	0.626242379
77011	115.94	72.92	0.628946007
121192	259	162.98	0.629266409
190818	692.55	432.89	0.625066782
134909	357.67	217.59	0.608354069
194634	741.59	451.53	0.608867433
304628	－	－	－

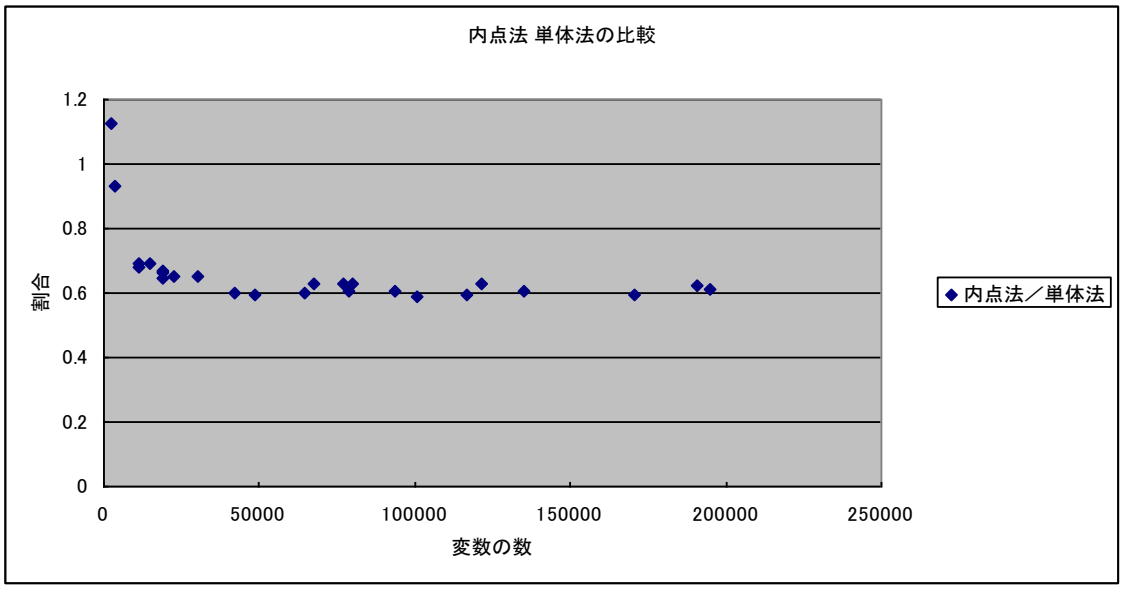
上表で － はメモリーオーバーを表している．

変数の数と計算時間のグラフ



上グラフを見ると内点法の方が計算時間が早いことが分かる．以下のグラフは内点法と単体法の計算時間の比をとったものである．

変数と計算時間のグラフ（内点法／単体法）



変数の数が増加しても，速度の比はおおよそ 0.6 をキープしている．この手の比較結果は問題依存によるところが多いので，今回の１つで単体法と内点法の比較とはいえないが，知見の一つしては役に立つだろうと考えている．

ここで紹介した計算時間は制約式の展開は含んでおらず，それを含めると計算時間はもっと大きいものであるため注意されたい．