LocalSolver7.0
使用手引書

MSI 株式会社

2017/6/1

目次

１．　LocalSolver とは

LocalSolver7.0 は、大規模最適化問題を実用時間内で効率よく求めることを目的とした新しい数理計画法システムである。

LocalSolver はフランスの 6 人の若手 OR 実践者が 10 年の歳月をかけて開発したものであり、All-In-One　Solver として、MIP（混合整数計画法）、CP(制約論理プログラミング)、NLP（非線形計画法）、LP（線形計画法）問題を解くことができる。　特に、MIP、CP では現実的に解けない大規模最適化問題に対して、メタヒューリスティクス（ローカルサーチ）解法を進化させた解法を持った、次世代の数理計画法システムである。

２．　LocalSolver の実行方法（稼働確認）

LocalSolver は PC,　Unix,　MAC のそれぞれで 32bits または 64bits モードで稼働する。

LocalSolver は C++で開発されており、実行プログラムを単体で動かすだけでなく、C++、Java、C#、Python のプログラムから呼ぶことができる。

LocalSolver はインタープリター型の実行プログラムを直接実行することができる。インタープリター型で LocalSolver を実行する場合には、最新の関数型プログラミングをベースとした LSP ファイル（xxx.lsp）を作成し、パラメータとして LSP ファイルを指定する必要がある。

また、LocalSolver が提供するクラスライブラリを使うことで、効率よく、C++, Java,　C#（.net）、python で、開発することが可能である。

本ドキュメントでは、PC（32bits 版）で DOS のコマンドプロンプトから LocalSolver を直接実行する例を説明する。

LocalSolver を実行するためには、Localsolver をインストールし、ライセンスを獲得する必要がある。その手順については、以下を参照されたい。

https://www.msi-jp.com/localsolver/download/

2.1　LSP ファイルの実行

インストールされた内容を示す。

localsolver_7_0　フォルダ
- ｜－　bin　　：実行プログラム localsolver.exe
- ｜－　docs　　：LSP, C++, C#, Java のクラスライブラリの説明
- ｜－　examples　　：例題集
- ｜－　include
- license.dat　　：自分の PC 用のライセンスキーに置き換える。

１）実行手順
- DOS コマンドプロンプトを立ち上げる。
- LSP ファイルが入っているフォルダにディレクトリを移す。
  例）**cd C:¥localsolver_7_0¥examples¥toy**
- LSP ファイル名を指定して、localsolver.exe を実行する。
  例）**localsolver toy.lsp lsTimeLimit=1**
  ※LSP ファイル名：toy.lsp。
  ※パラメタ lsTimeLimit は実行時間を 1 秒と指定。

２）LSP ファイルの例

ここでは、examples¥toy の LSP ファイル:toy.lsp を示す。
Toy モデルは、ナップサック問題である。
品物が、８品あり、それぞれの重さと価値以下に定義する。
重さ：10、60、30、40、30、20、20、2　　kg
価値：1、10、15、40、60、90、100、15　　円
ナップサックには最大 102kg まで品物を入れることができ、価値が最大になるよう、どの品物を選べばよいか、その時の価値はいくらになるかが問題である。
LSP による定式化を以下に示す（toy.lsp ファイルの内容）。

```
/********** toy.lsp **********/
function model()
{
  // 0-1 decisions
  x_0 <- bool(); x_1 <- bool(); x_2 <- bool(); x_3 <- bool();
```

```
  x_4 <- bool(); x_5 <- bool(); x_6 <- bool(); x_7 <- bool();
    // weight constraint
  knapsackWeight <- 10*x_0 + 60*x_1 + 30*x_2 + 40*x_3 + 30*x_4 + 20*x_5 + 20*x_6 + 2*x_7;
  constraint knapsackWeight <= 102;
    // maximize value
  knapsackValue <- 1*x_0 + 10*x_1 + 15*x_2 + 40*x_3 + 60*x_4 + 90*x_5 + 100*x_6 + 15*x_7;
  maximize knapsackValue;
to
```

※赤字は予約語。この例では、model()　関数で、データと制約式、目的関数を定義してある。
※bool()が意志決定変数を意味し、bool()で定義された変数の０と１の値の組合せを高速に評価することで、最適解を求める。

2.2　実行結果

2.1 で示した toy モデル（toy.lsp ファイル）の実行結果を以下に示す。

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:¥Users¥Miyazaki-MSI>cd C:¥localsolver_7_0with64¥examples¥toy

C:¥localsolver_7_0with64¥examples¥toy>localsolver toy.lsp lsTimeLimit=1
LocalSolver 7.0 (Win64, build 20170502)
Copyright (C) 2017 Innovation 24, Aix-Marseille University, CNRS.
All rights reserved. See LocalSolver Terms and Conditions for details.
Load toy.lsp...
Run model...
Preprocess model 100% ...
Close model 100% ...
Run param...
Run solver...
Initialize threads 100% ...
Push initial solutions 100% ...

Model:
  expressions = 38, operands = 50
  decisions = 8 (bool = 8, int = 0, float = 0, list = 0),
  constraints = 1, objectives = 1, constants  = 11
  Preprocessing transformed 11 expressions

Param:
  time limit = 1 sec, no iteration limit
```

```
   seed = 0, nb threads = 2, annealing level = 1

Objectives:
   Obj 0: maximize, bound = 331

Phases:
   Phase 0: time limit = 1 sec, no iteration limit, optimized objective = 0


Phase 0:
[    0 sec,          0 itr] : obj =              280
[    1 sec,    137637 itr] : obj =              280
[    1 sec,    137637 itr] : obj =              280

137637 iterations, 275164 moves performed in 1 seconds
Feasible solution: obj =              280

C:¥localsolver_7_0with64¥examples¥toy>
```

※ちなみに答えは、
x_2(30,15)， x_4(30,60)， x_5(20,90)， x_6(20,100)， x_7(2,15)の５個であり、重さの合計は 102kg、価値の合計は 280 円となる。
添付資料４に toy2.lsp として、output function を使って答えを出力した例を示す。

実行ログには、以下が示される。

- [1 sec, 137637 itr] :経過時間、イタレーション数（解の数）
- obj = 280 :目的関数値
- 137637 iterations, 275164 moves performed in 1 seconds
- Feasible solution: obj = 280
- 275164 moves :試行回数（マルチスレッド合計のイタレーション数）
-

時間指定等で打ち切られた場合または、実行不可能解もしくは最適解を判定した場合に、以下の３つの状態を出力する。

- infeasible ： 実行不可能解
- feasible ：実行可能状態
- optimal ：最適解

３．　LocalSolver による定式化

LocalSolver の定式化は意志決定変数を定義することから始まる。

意志決定変数には(bool)として定義した 0-1 変数、上下限を持つ整数変数(int)、上下限を持つ実数変数(float)、(list) として定義するセットの組合せからなり、意思決定変数の値を変化させ、超高速に解の探索を試行することで、大規模最適化問題を実用的に解くことが基本の考え方である。

bool 変数の数がたとえ 1000 万変数を超えても実用的な意味で解を求めることができる。bool 変数で定義した意志決定変数の組合せとそのもとで実行可能となる bool 変数だけを変化させ解を探索していくため、LocalSolver 用に定式化するためには、以下の点に注意する必要がある。

ナップサック問題のように条件に合う品物を選ぶのが目的であれば、品物を選ぶか選ばないかを bool 変数として定義すれば良い。ある品物が選ばれた場合、bool 変数を使って品物の重さ及び価値を直接計算できるため、bool 変数の組合せで制約条件、目的関数を評価することができる。

LocalSolver7.0 では、従来の MIP 問題を扱うことができ、最適化計算途中にマテリアルバランスを求め、どこで何をいくつ作る／使うかを答えとして求めることができる。この場合、拠点配置や設備投資を決める固定費問題として、生産数量等の変数を float 変数として定義し、使う/使わないを決める意志決定変数を、bool 変数として定義すればよい（従来は bigM を使用して関連付ける必要があったが技巧的な定式化は不要である)。

※ただし、既存の MIP 問題の定式化では、解探索が必ずしも速くならないため、LocalSolver に適した問題定義が重要である。

3.1　LSP モデリングの考え方

以下の手順でモデリングを行う。

１） 意志決定変数(bool 変数及び float 変数等)を定義する。

２） bool 変数が選ばれた場合に、制約、目的関数が必要とする数量（生産量等）を可変としたい場合には、float 変数で定義することができる。

３）bool 変数及び float 変数を使い、制約条件、目的関数を定義する。

この時、MIP のように、線形制約にこだわる必要はなく、制約条件、目的関数ともに、非線形表現が可能である。

3.2　LSP での bool 変数の定義

以下に典型的な問題毎に bool 変数の定義イメージを示す。

- ナップサック問題：Xp　　（p ：品物）
- ルート選択問題：　Xr　　(r：ルート)
- 裁断計画問題：　　Xp,q　（p ：裁断パターン、 q ：パターンの使用回数）
- 人員配置問題：　　Xp,t,j　(p：人員、 t ：時間、 j ：ジョブ）
- 車両投入計画：　　Xc,p　（c ：車両、 p ：ポジション（順番））
- SCM：　　Xt,i,j,k,p (t：期、i：工場、j：ライン、k：倉庫、p：製品)
- スケジューリング：Xt,i,j,p　(t：時間等、i：工場、j：ライン、p：製品等)

,

LocalSolver では数百万以上の bool 変数を定義することができ、MIP の定式化と違い、オーダ単位に意思決定変数(bool)を定義することで、より、自然な形での定式化が可能となる。

LocalSolver7.0 は、最初の事前解析で実行可能解を求めるため、利用者が実行不可能性またはび実行可能性を考慮する必要はない。

3.3　LSP モデル

LSP モデルは、以下の要素から構成される。

- 意志決定変数：bool()、float（下限値、上限値）、int（下限値、上限値）、list()
- 副生変数:任意の変数であり、プログラミングをわかりやすくすることができる。変数の定義には、<－　を使用する。
- 制約：constraint（予約語）で、制約条件を定義する。
  constraint 制約式で定義された値が実行可能性の判定で使用される。
- 目的関数：minimize（予約語）または maximaize（予約語）で目的関数を定義する。目的関数は複数定義可能であり、定義された順番に最適化を行う。目的計画法として利用可能である。

４．　LSP 言語

LSP 言語は、最適化問題をモデル化し、モデルの検証及び解の検証を行うフェーズでの試行錯誤を行うのに最適な環境を提供することを目的として開発されている。

LSP言語は、最新の関数型プログラミング言語である。関数型プログラミング言語の特長は、型推論を備えた言語であるため、JavaやC言語と異なり、コンパイラが自動的にデータの種類を推定するため、データの種類（型）をプログラマが指定する必要がない。その結果、プログラムの記述はRubyなど軽量言語のように簡潔である。
軽量言語では実現できないコンパイラによるプログラムのチェックが可能になる点にある。

LSP言語の特徴は以下：
－迅速に開発できる（開発生産性が良い。従来に比べて、1/5から1/2の開発量）
－バグを抑えやすい（コンパイラが型の間違い等を自動的にチェックする）
－アプリケーションの性能を向上させやすい
－簡潔かつシンプルなモデリング言語（できるだけ省略できるよう設計）
※大規模問題でも制約条件及びデータがそろっていれば、1 日でモデリングと実行が可能である。
－作成(修正)←→実行が同時にできる（一つはエディタ、もう一つは DOS コマンドプロンプトの二つのウインドウを操作しながら開発が可能である。
－目的計画法のように目的を段階的に設定することができるため、モデルの開発及び解の検証を段階的に行うことができる。

添付資料３に lsp 言語と C++、Java、C#で記述した例を示す。

4.1　Function

LSP はメインプログラムがなく、以下の５つの基本的なファンクションからなる。
function model()は必須であるが、その他は必要に応じて使用すればよい。
また、以下の基本的なファンクションを使用することができる。

- `input`: for declaring your data or reading them from files.
- `model`: for declaring your optimization model.
- `param`: for parameterizing the local-search solver before running.

- `display`: for displaying some info in console or in some files during the resolution.
- `output`: for writing results in console or in some files, once the resolution is finished.

4.2 演算子

LSP モデルの中で、自由に使用できる。とくに、目的関数、制約条件の記述に利用でき、非線形制約、非線形目的関数として利用可能である。

演算子には、以下の種類がある。詳細は一覧表を参照されたい。

- 算術演算子（sum、min、max、sin、cos、log、exp 等）
- 論理演算子（not、and、or、xor）
- 関係演算子（==、 !=、<=、>=、<、>）
- 複合演算子（if、array+at）

4.3 言語機能

1）変数定義

変数の定義の例を示す。以下はすべて有効である。

```
a = true;   // a = 1
b = 9;      // b = 9

c = a + b;  // c = 10
c = a * b;  // c = 9

c = a == b; // c = 0
c = a < b;  // c = 1
```

2）配列定義

LocalSolver の配列は　map　で定義することができる。

Map は、値とキーを併せ持ったデータ構造になっている。 キーは、整数であり、かならずしも連続的である必要はない。値は、どんなタイプでもで可能であり、 キーに対応させるために、[ブラケット]表記法を用いる。

```
a = map("z", 9); // a[0] = "z", a[1] = 9
```

```
a = {"z", 9};    // a[0] = "z", a[1] = 9

a["a"] = "abc";  // a[0] = "z", a[1] = 9, a["a"] = abc
```

3）条件判定

条件判定は、if 文を使用する。記述形式は、以下：

if (C) S_true; else S_false;

または、？・：で簡潔に記述することもできる。

```
if (1 < 2) c = 3; else c = 4;
c = 1 < 2 ? 3 : 4;

if (0) c = "ok";
if (true) c = "ok";
if (2) c = "error"; // ERROR: invalid condition

c = 0 * 9; // c = 0
if (c) {
   a = "L";
   b = 0;
} else { // executed block
   a = "S";
   b = 1;
}
```

4）繰り返し

繰り返しには、while と for がある。

While は、以下で記述する。C が真である限り、S が実行される。

```
do S; while (C);
```

for は、以下で記述する。 v が V にある限り、S が実行される。

```
for [v in V] S;
```

また, キーと値がセットの場合には、以下で記述する。

```
for [k, v in M] S;
```

```
for [i in 0..2] a[i] = i + 1; // a[0] = 1, a[1] = 2, a[2] = 3
```

```
s = 0; for [v in a] s = s + v; // s = 6
s = 0; for [k,v in a] s = s + k + v; // s = 9

for[i in 0..9]
   for [j in i+1..9]
      for [k in j+2..9]
         a[i][j][k] = i + j + k;

for[i in 0..9][j in i+1..9][k in j+2..9] // compact
   a[i][j][k] = i + j + k;

for[i in 0..9][j in i+1..9][k in j+2..9]
{
   a[i][j][k] = i + j + k;
   b[i][j][k] = i * j * k;
}
```

5）繰り返し演算

繰り返し演算は、以下で記述する。

```
for [v in V] a[v] = f(v);
```

LSP では、以下の省略形で記述可能である。

```
a[v in V] = f(v);
```

```
for[i in 0..9][j in i+1..9][k in j+2..9]
   a[i][j][k] = i + j + k;

a[i in 0..9][j in i+1..9][k in j+2..9] = i + j + k;

x[i in 0..n-1][j in 0..m-1] <- bool();.
```

6）関数

LSP では、任意に関数を定義できる。関数の値は、0(false)または1(true)でも良いし、数値でも良い。LSP プログラムは通常、function 間で共通の変数定義(global)になっているため、function 内でローカルに使用したい場合には、local の宣言子でローカル変数であることを定義する必要がある。

```
function isEven(v) {
  if (v % 2 == 0) return true;
  else return false;
}

function computeSumOfEvenNumbers(a,b) {
    local total = 0;
    for [v in a..b : isEven(v)]
        total = total + v;
    return total;
}
```

4.4　主なエラーメッセージ

コマンド・ラインで絶対に必要な引数は、lsp ファイルの名前です。もし lsp ファイルが 利用できないならば、エラーを出す。また、 コマンド・ラインの他の全ての引数（パラメータ等）は、フォーマット identifier=value を持たなければならない。

・<f> doesn't exist or is not accessible. // LSP file
・Invalid argument format for <arg>. Expected format : identifier=value.

LSP は型を強く意識した言語であり、関数のパラメータには正しい型が必要である。

・Function <f> cannot handle argument of type <t>. Argument of type <t2> is expected.
・Function <f> takes <x> argument(s) but <y> were provided.
・Function <f> : <T> expression expected for argument <i>.

同様に、型チェックで不適切な型の場合には、エラーメッセージを出力する。

・Cannot apply <opName> operator on type <T>.
・Cannot apply <opName> operator between types <T1> and <T2>
・Cannot cast <T1> to <T2>.
・Cannot apply ternary operator '?:' on given operators : incorrect argument type.
・Cannot cast 'nil' to <T>. A variable or a map element may not be assigned.

いくつかの関数は引数を持つ。もし、引数の数が合わない場合には、エラーを出力する。：

・Function <f> takes at least <x> argument(s) but <y> were provided.
・Function <f> takes at most <x> argument(s) but <y> were provided.

関数を使う時、関数が未定義であれば、エラーメッセージを出力する。また、既存の関数と同じ名前の関数を定義するのもエラーである。
変数にかんしては、自由に再定義（再利用）可能である。ただし、局所変数だけは、同じ名前で二回使うことはできない。 もし、変数が値を持たない場合は、nil の値を持つ。

・Function <f> already defined.
・Function <f> undefined.
・Variable <name> already defined.

Input/output 関数は、指定されたファイルの入出力チェックを行う。

・File <f> cannot be opened.
・Cannot read from file <f>.
・Cannot write to file <f>.

数字または文字列を入力時に、プログラムとデータが一致しない（データ数、タイプ等が一致しない）場合およびファイルの最後まで読んだ場合にはエラーを出力する。

・Cannot convert the current token to int.
・Cannot convert the current token to double.
・End of file: no more line to read from file <f>.
・End of file reached.

文字列の操作では、文字列が空でないこと及びインデックスが許容範囲内であることが必要である。

・The given index for substring is out of range. Min value: 0, Max value: <len>.
・Number of characters for substring must be greater than 0.
・Search string is empty.

マップの制限として二つある。

−キーは整数または文字であること

−イタレーション中（連続して探索計算している間）は、マップを変更してはならない:

・'nil' provided as key for a map. The key variable may not be assigned.
・Only types 'string' and 'int' are allowed for keys in maps.
・Cannot iterate on a modified map.

LSP モデルに対してパラメータで数値を指定する場合には、許容範囲の数値でなければならない。

・The objective bound must be an integer, a double or a boolean for objective <objIndex>
・The objective bound must be an integer or a boolean for objective <objIndex>
・The number of threads cannot exceed 1024.
・The annealing level size must be an integer between 0 and 9.
・Advanced parameter <key> does not exist.

モデルで式または変数を表現する場合には、<- で宣言する必要である。.局所変数を <-を使って宣言することは出来ない。
モデルには、必ず目的関数がなければならない。また、目的関数は数式で表現する必要がある。
マップ等は使用できなう。
制約式は、バイナリ表現でなければならない。

・Cannot assign localsolver expressions to local variables.
・At least one objective is required in the model.
・Only boolean expressions can be constrained.
・Only expressions with a value can be added in the objectives list.

setValue 関数は、意志決定変数（bool 変数）にのみ初期値を与えることができる。
.
・The only allowed values are 0 or 1.

x <- a[y] の数式表現では、マップとしてゼロから連続した整数キーが必要となる。また、バリューとして、キーの数ぶんだけ、数値データまたはLS 表現が必要である。

・All keys must be integers. Type found: <T>
・Values must be integers, booleans or expressions. Type found: <T>
・The first key must be 0. Key found: <key>
・Keys are not in a continuous range. Next key expected <key1>. Key found: <key2>.

探索の間に、　変数値が制約を満足しない場合が発生することに注意されたい。
例えて言えば、実行可能状態の時に、実行可能解を探索中に起きるケースがある。
ゼロ割や配列オーバーフローが起きた時であり、割算の分母がゼロまたはインデックスが範囲外になったことを意味する。 z <- x/y のような場合には、 z<-x/max(1,y)と表現することが望ましい。

・Division by zero.
・Index out of bounds for 'at' operator (index: <indexId>, array size: <n>

以上

【付録１】演算子一覧表

Operator	Description	Type	Arity	Symb	
Decisional	bool	Boolean decision: decision variable with domain {0,1}.	bool	0	
Arithmetic	sum	Sum of all operands.	double*	$n > 0$	+
	prod	Product of all operands.	double*	$n > 0$	*
	min	Minimum of all operands.	double*	$n > 0$	
	max	Maximum of all operands.	double*	$n > 0$	
	div	Division of the first operand by the second one.	double*	2	/
	mod	Modulo: mod(a,b) = r such that $a = q*b + r$ with q, r integers and $\|r\| < b$.	int	2	%
	abs	Asolute value: abs(e) = e if $e >= 0$, $-e$ otherwise.	double*	1	
	dist	Distance: dist(a,b) = abs($a-b$).	double*	2	
	sqrt	Square root.	double	1	
	cos	Cosine.	double	1	
	sin	Sine.	double	1	
	tan	Tangent.	double	1	
	log	Natural logarithm.	double	1	
	exp	Exponential function.	double	1	

	pow	Power: pow(*a*,*b*) is equal to the value of *a* raised to the power of *b*.	double	2	
	ceil	Ceil: round to the smallest following integer.	int	1	
	floor	Floor: round to the largest previous integer.	int	1	
	round	Round to the nearest integer: round(*x*) = floor(*x*+0.5).	int	1	
Logical	**not**	Not: not(*e*) = 1 − *e*.	bool	1	!
	and	And: equal to 1 if all operands are 1, and 0 otherwise.	bool	*n* > 0	&&
	or	Or: equal to 0 if all operands are 0, and 1 otherwise.	bool	*n* > 0	||
	xor	Exclusive or: equal to 0 if the number of operands with value 1 is even, and 1 otherwise.	bool	*n* > 0	
Relational	**eq**	Equal to: eq(*a*,*b*) = 1 if a = b, and 0 otherwise.	bool	2	==
	neq	Not equal to: neq(*a*,*b*) = 1 if a != b, and 0 otherwise.	bool	2	!=
	geq	Greater than or equal to: geq(*a*,*b*) = 1 if a >= b, and 0 otherwise.	bool	2	>=
	leq	Lower than or equal to: leq(*a*,*b*) = 1 if a <= b, and 0 otherwise.	bool	2	<=
	gt	Strictly greater than : gt(*a*,*b*) = 1 if a > b, and 0 otherwise.	bool	2	>
	lt	Strictly lower than: lt(*a*,*b*) = 1 if a < b, and 0 otherwise.	bool	2	<
Conditional	**iif**	Ternary conditional operator: iif(*a*,*b*,*c*) = *b* if a is equal to 1, and *c* otherwise.	int*	3	? :
	at	Access to an array: *T*[*i*] returns the *i*th value in array *T*.	int	2	[ ]

【付録2】予約語一覧

Modeling & Solving Parameter

- `lsTimeLimit = {10, 50};` Spends 10 (resp. 50) sec to optimize objective 0 (resp. 1).
- `lsTimeLimit = 60;` Corresponds to `lsTimeLimit = {0,..., 0, 60}`.
- `lsIterationLimit = {1000, 5000};` Spends 1000 (resp. 5000) iterations to optimize objective 0 (resp. 1).
- `lsIterationLimit = 6000;` Corresponds to `lsIterationLimit= {0,..., ,6000}`.
- `lsTimeBetweenDisplays = 5;` Displays info about the search every 5 sec (default: 1).
- `lsSeed = 9;` Sets pseudo-random number generator seed to 9 (default: 0).
- `lsNbThreads = 4;` Parallelizes the search over 4 threads (default: 2).
- `lsAnnealingLevel = 9;` Sets simulated annealing level to 9 (no annealing: 0, default:1).
- `lsVerbosity = 1;` Sets verbosity to 1 (no display: 0, default: 1).

Functions

Input & Output Library

- `f = openRead("data.in");` Opens file "data.in" in reading mode.
- `f = openWrite("data.out");` Opens file "data.out" in writing mode.
- `f = openAppend("data.out");` Opens file "data.out" in append mode.
- `close(f);` Closes the file.
- `eof(f)` Returns true if the end of file is reached.
- `i = readInt();` Prompt the user for an int (in the console standard input).
- `i = readInt(f);` Reads the next int parsed in file.
- `i = readDouble();` Prompt the user for a floating-point number (in the console standard input).
- `i = readDouble(f);` Reads the next floating-point number parsed in file.
- `s = readln();` Prompt the user for a line (in the console standard input).
- `s = readln(f);` Reads the next line of file.
- `s = readString(f);` Reads the next string parsed in file.
- `print("s = " + s + "¥n");` Prints the string in console.
- `print(f, "s = " + s + "¥n");` Prints the string in file.
- `println("s = " + s);` Prints the string followed by a line feed in console.
- `println(f, "s = " + s);` Prints the string followed by a line feed in file.
- `error(msg);` Prints an error message and exits.

Map Library

- `m = map(); m = {};` Creates an empty map.
- `m = map(9, "abc"); m = {1, "abc"};` Creates a map containing values 9, "abc" at keys 0, 1 respectively.
- `nbElems = count(m);` Counts the number of values in the map.
- `elems = values(m);` Returns the values of the map as a map.
- `indices = keys(m);` Returns the keys of the map as a map.
- `add(m, 123);` Adds 123 in the map with key equals to the largest integer key plus one.

String Library

- `i = toInt("123");` Converts the string into the corresponding integer (or throws an error if not possible).
- `i = toDouble("123.45");` Converts the string into the corresponding floating-point number(or throws an error if not possible).
- `m = split("a::b::c::d", "::");` Splits string "a::b::c::d" into substrings (as a map) according to the

separator "::".

- `s = trim("  abcd  ");` Removes white spaces at the beginning and at the end of the string.
- `len = length("abcd");` Returns the length of a string.
- `s = substring("abcd",1,2);` Returns a new string that is a substring of this string.There are two versions of this function: The first one takes two arguments : the string, and the start index of the substring. The second one takes 3 arguments : the string, the start index and the length of the substring.
- `b = startsWith("abcd","ab");` Returns true if the first argument starts with the specified prefix given as a second argument.If the second argument is the empty string, returns true.
- `b = endsWith("abcd","cd");` Returns true if the first argument ends with the specified suffix given as a second argument.
- `s = lowerCase("ABCD");` Returns a new string converted to lower case.
- `s = upperCase("abcd");` Returns a new string converted to upper case.
- `s = replace("abcd","bc","x");` Replaces each substring of a string that matches the literal target string with the specified literal replacement string. The replacement

proceeds from the beginning of the string to the end, for example, replacing "aa" with "b" in the string "aaaaa" will result in "bba" rather than "abb".This function takes 3 arguments :subject string, searched sequence and replace sequence.

Modeling & Solving Library

- `getObjectiveBound(1);` Gets the bound of objective (with index) 1.
- `setObjectiveBound(1, 9999);` Sets the bound of objective 1 to 9999.
- `v = getValue(x);` Gets the value of modeling expression `x` in the best solution found by the solver.
- `setValue(x, 1);` Sets the value of `x` to 1 in the initial solution (or throws an error if `x` is not a decision).

【付録３】　言語比較

モデル名	lsp	C++	lsp 比	Java	lsp 比	C#	lsp 比	備考
toy	17	49	2.9	46	2.7	48	2.8	ナップサック問題（トイモデル）
car_sequencing	77	211	2.7	211	2.7	246	3.2	車両投入順序問題
knasack	44	126	2.9	128	2.9	144	3.3	ナップサック問題
maxcut	44	138	3.1	137	3.1	149	3.4	裁断計画問題
multiobj_knapsack	93	172	1.8	182	2.0	191	2.1	ナップサック問題（非線形最適化）
pmedian	66	154	2.3	154	2.3	169	2.6	OR-LIB。２点間の輸送コスト最小
平均比較	1.0		2.6		2.6		2.9	
socialgolfer	69							CSPLIB, problem 010
steel_mill_slab_design	99							CSPLIB, problem 038
google_machine	231							Google 問題（Google がスポンサ）
table_layout	243							表配置問題
nurse_rostering	505							看護師スケジューリング

【付録４】　サンプル２（lsp と実行結果）

付録4．１　lspプログラム（toy モデルに関数：output を追加）

```
/********** toy2.lsp **********/
function model()
{
   nbProducts = 8;
   value = {1,10,15,40,60,90,100,15};
// 0-1 decisions
   x[i in 0..nbProducts-1] <- bool();
   // weight constraint
   knapsackWeight <- 10*x[0] + 60*x[1] + 30*x[2] + 40*x[3] + 30*x[4] + 20*x[5] +
20*x[6] + 2*x[7];
   constraint knapsackWeight <= 102;
   // maximize value
   knapsackValue <- 1*x[0] + 10*x[1] + 15*x[2] + 40*x[3] + 60*x[4] + 90*x[5] +
100*x[6] + 15*x[7];
   maximize knapsackValue;
}

function output()
{
   println("Selected Products:");
   for [i in 0..nbProducts-1 : getValue(x[i]) == 1]
      println("#"+i+" ("+value[i]+")");
}
```

付録4．２　lsp実行結果（toy モデルに関数：output を追加）

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.    All rights reserved.

C:¥Users¥Miyazaki-MSI>cd C:¥localsolver_7_0with64¥examples¥toy

C:¥localsolver_7_0with64¥examples¥toy>localsolver toy2.lsp lsTimeLimit=1
LocalSolver 7.0 (Win64, build 20170502)
Copyright (C) 2017 Innovation 24, Aix-Marseille University, CNRS.
All rights reserved. See LocalSolver Terms and Conditions for details.
Load toy2.lsp...
Run model...
Preprocess model 100% ...
Close model 100% ...
Run solver...
Initialize threads 100% ...
Push initial solutions 100% ...

Model:
   expressions = 38, operands = 50
   decisions = 8 (bool = 8, int = 0, float = 0, list = 0),
   constraints = 1, objectives = 1, constants    = 11
   Preprocessing transformed 11 expressions

Param:
   time limit = 1 sec, no iteration limit
   seed = 0, nb threads = 2, annealing level = 1

Objectives:
   Obj 0: maximize, bound = 331

Phases:
   Phase 0: time limit = 1 sec, no iteration limit, optimized objective = 0
```

Phase 0:

[0 sec, 0 itr] : obj = 280

[1 sec, 75777 itr] : obj = 280

[1 sec, 75777 itr] : obj = 280

75777 iterations, 151597 moves performed in 1 seconds

Feasible solution: obj = 280

Run output...

Selected Products:

#2 (15)

#4 (60)

#5 (90)

#6 (100)

#7 (15)

C:¥localsolver_7_0with64¥examples¥toy>

【付録5】 BNF Syntax（バッカス・ナウア記法）

BNF の表記は次のような導出規則の集合である。

```
<symbol> ::= <expression with symbols>
```

左辺の<symbol>は単一の記号である。また、<expression with symbols> は記号列、または選択を表すバーティカルバー「|」で区切られた記号列であり、左辺の &code><symbol> の置換となるものを表している。なお、導出規則で使用された記号は「非終端記号」と「終端記号」に分類される。導出規則群の左辺に現れた記号は「非終端記号」と呼ばれ、いずれの導出規則の左辺にも現れなかった記号は「終端記号」と呼ばれる。

以下に LSP 言語の BNF Syntax を示す。.
※＜＞::＝＜＞を省略してある。

```
expression
    : arithm_expression
    | table_expression
    | range_expression
    ;
arithm_expression
    : primary_expression
    | lambda_expression
    | arithm_expression '||' arithm_expression
    | arithm_expression '&&' arithm_expression
    | arithm_expression '==' arithm_expression
    | arithm_expression '!=' arithm_expression
    | arithm_expression 'is' arithm_expression
    | arithm_expression 'is' 'nil'
    | arithm_expression 'is' 'int'
    | arithm_expression 'is' 'double'
    | arithm_expression '<' arithm_expression
    | arithm_expression '>' arithm_expression
    | arithm_expression '<=' arithm_expression
    | arithm_expression '>=' arithm_expression
    | arithm_expression '+' arithm_expression
    | arithm_expression '-' arithm_expression
```

```
    | arithm_expression '*' arithm_expression
    | arithm_expression '/' arithm_expression
    | arithm_expression '%' arithm_expression
    | arithm_expression '?' arithm_expression ':' arithm_expression
    | '+' arithm_expression
    | '-' arithm_expression
    | '!' arithm_expression
    | 'typeof' arithm_expression
    ;
primary_expression
    : identifier
    | 'true'
    | 'false'
    | 'nan'
    | 'inf'
    | 'nil'
    | string
    | integer
    | double
    | primary_expression '[' expression ']'
    | primary_expression '.' identifier
    | function_call
    | '(' expression ')'
    ;
lambda_expression
    : identifier '=>' block_statement
    | '(' function_identifier_list ')' '=>' block_statement
    | '(' ')' '=>' block_statement
    |  identifier '=>' arithm_expression
    | '(' function_identifier_list ')' '=>' arithm_expression
    | '(' ')' '=>' arithm_expression
    | 'function' '(' function_identifier_list ')' block_statement
    | 'function' '(' ')' block_statement
    ;
range_expression
    : arithm_expression '..' arithm_expression
```

```
    ;
table_expression
    : '{' '}'
    | '{' table_list '}'
    ;
table_list
    : expression
    | table_key '=' expression
    | table_key ':' expression
    | table_list ',' expression
    | table_list ',' table_key '=' expression
    | table_list ',' table_key ':' expression
    ;
table_key
    : string
    | identifier
    | integer
    | '-' integer
    ;
function_call
    : primary_expression '(' ')'
    | primary_expression '(' function_argument_list ')'
    | primary_expression variadic_compositor_list '('
function_argument_list ')'
    ;
function_argument_list
    : expression
    | function_argument_list ',' expression
    ;
variadic_compositor_list
    : '[' filter_iterator ']'
    | variadic_compositor_list '[' filter_iterator ']'
    ;
filter_iterator
    : identifier 'in' expression ':' expression
    | identifier ',' identifier 'in' expression ':' expression
```

```
    | identifier 'in' expression
    | identifier ',' identifier 'in' expression
    ;
statement
    : block_statement
    | assignment_statement
    | local_assignment_statement
    | local_statement
    | if_else_statement
    | for_statement
    | while_statement
    | dowhile_statement
    | continue_statement
    | break_statement
    | modifier_statement
    | throw_statement
    | trycatch_statement
    | function_call_statement
    | return_statement
    | ';'
    ;
block_statement
    : '{' '}'
    | '{' statement_list '}'
    ;
statement_list
    : statement
    | statement_list statement
    ;
assignment_statement
    : identifier assignment_operator expression ';'
    | identifier assignment_compositor_list assignment_operator
expression ';'
    ;
assignment_operator
    : '='
```

```
    | '<-'
    | '+='
    | '-='
    | '/='
    | '*='
    | '%='
    ;
assignment_compositor_list
    : assignment_compositor
    | assignment_compositor_list assignment_compositor
    ;
assignment_compositor
    : '[' filter_iterator ']'
    | '[' arithm_expression ']'
    | '[' range_expression ']'
    | '.' identifier
    ;
local_assignment_statement
    : 'local' identifier local_assignment_operator expression ';'
    | 'local' identifier assignment_compositor_list
local_assignment_operator expression ';'
    ;
local_assignment_operator
    : '='
    | '<-'
    ;
local_statement
    : 'local' identifier ';'
    ;
if_else_statement
    : 'if' '(' expression ')'
    | 'if' '(' expression ')' 'else' statement
    ;
for_statement
    : 'for' for_compositor_list statement
    ;
```

```
for_compositor_list
    : for_compositor
    | for_compositor_list for_compositor
    ;
for_compositor
    : '[' filter_iterator ']'
    | '[' range_expression ']'
    ;
while_statement
    : 'while' '(' expression ')' statement
    ;
dowhile_statement
    : 'do' statement 'while' '(' expression ')' ';'
    ;
continue_statement
    : 'continue' ';'
    ;
break_statement
    : 'break' ';'
    ;
modifier_statement
    : modifier expression ';'
    ;
modifier
    : 'minimize'
    | 'maximize'
    | 'constraint'
    ;
throw_statement
    : 'throw' expression ';'
    | 'throw' ';'
    ;

trycatch_statement
    : 'try' statement 'catch' '(' identifier ')' statement
    ;
```

```
function_call_statement
    : function_call ';'
    ;
return_statement
    : 'return' ';'
    | 'return' expression ';'
    ;
function_list
    : function
    | function_list function
    ;
function
    : 'function' identifier '(' function_identifier_list ')'
block_statement
    | 'function' identifier '(' ')' block_statement
    ;
function_identifier_list
    : identifier
    | function_identifier_list ',' identifier
    ;
use_section
    : use_statement
    | use_section use_statement
    ;
use_statement
    :  identifier ';'
    ;
start
    : TOKEN_END
    | function_list TOKEN_END
    | use_section TOKEN_END
    | use_section function_list TOKEN_END
    ;
```

以上

【付録6】 LocalSolver 新機能

LocalSolver6.5 の新機能概要を以下に紹介する。

付録 6.1 Setting an initial solution

付録 6.2 Infeasibility and inconsistency Analysis

付録 6.3 At operator function

付録 6.4 Callbackfunctions

付録 6.5 Black-box Optimization

付録 6.6 List variables

付録 6.7 Function Operator (delegates)

付録 6.8 Piecewise operator

6.1 Setting an initial solution

LocalSolver does not need a starting solution to launch its algorithms.

However in some cases you may want to force LocalSolver to start from a specific solution. For instance a planning system may consist in reoptimizing every morning the current planning (inserting new tasks and taking into account updated deadlines). In such a case passing an intial solution as input is natural.

Such an intialization will be achieved in LocalSolver by setting the value of decision variables. For numeric decision variables (boolean, integers and floats) it is done with the value attribute in the LSP modeler. For collection decision variable (lists) we use the add and clear functions on this value:

```
function param() {
    // with x an int, y a float and z a list
    x.value = 3;
    y.value = 4.3;
    z.value.clear();
    z.value.add(2);
    z.value.add(0);
}
```

Note that only decision variables can be initialized: setting the value of any other expression will throw an exception. Besides, it is not necessary to set the values of all decision variables. On the contrary in can make sense to set the values of some of the decision variables, while relyling on LocalSolver to initialize other values. It is also possible to intialize values to an infeasible solution, that is to say a solution violating some of the constraints. In this case, LocalSolver will start from this infeasible solution and quickly move to a feasible solution. The only requirement is that a decision variable cannot be given a value outside of its domain. For instance an integer decision defined as int(3,10) cannot be given value 15 and a list cannot be initialized to a collection with duplicated values.

In the APIs, the principle is the same.

Setting the value of a numeric expression is done with setIntValue or setValue for int and boolean decisions or with setDoubleValue for float decisions. Lists are modified with add and clear.

```
// With ls a LocalSolver object
LSSolution sol = ls.getSolution();
LSExpression intExpr = ls.getModel().getExpression("x");
LSExpression dblExpr = ls.getModel().getExpression("y");
LSExpression listExpr = ls.getModel().getExpression("z");

sol.setValue(intExpr, 12ll);
intExpr.setValue(12ll);

sol.setIntValue(intExpr, 12ll);
intExpr.setIntValue(12ll);

sol.setDoubleValue(dblExpr, 4.8);
dblExpr.setDoubleValue(4.8);

LSCollection col = listExpr.getCollectionValue();
col.clear();
col.add(2);
col.add(0);
col.add(3);
```

6.2 Infeasibility and inconsistency　Analysis

When submitting a model to LocalSolver (calling method solve), the expected result is to obtain a feasible solution, and even sometimes an optimal solution. However in some cases the returned solution can be infeasible in the sense that the current assignement of values to variables violates some of the constraints of the problem. Two solution status (see getSolutionStatus) are defined for these infeasibilty cases:
・infeasible means that no feasible solution was found to the submitted problem but it could not be proven that no such solution exists. Maybe running a longer search would have produced a feasible solution.
・inconsistent means that the solver was able to prove that no feasible solution exists. In this case, LocalSolver offers a functionality for analyzing the causes of this inconsistency.

□Note
Note that both status infeasible and inconsistent can also be encountered on problems where no constraint was defined. Indeed some LocalSolver expressions induce an implicit constraint on their operands. For instance sqrt(x) implicitly requires that x takes a non-negative value.

Analyzing inconsistencies

Analyzing an inconsistent model amounts to identifying a relatively small inconsistent subproblem. Such a subproblem or inconsistency core is said to be irreductible if it contains no smaller inconsistent subproblem.

The function computeInconsistency computes such a core that is to say a set of expressions (named causes) such that the problem restricted to these expressions and their descendents is inconsistent. Calling this function requires the model to be closed and the solver to be stopped. This inconsistency core is returned as an LSInconsistency object. This object can be printed in a readable form so that the user can easily spot the origin of the inconsistency. It also allows scanning the set of identified causes.

For example, the following model is inconsistent because limiting the cube of y to 250 prevents y from taking values larger than 6, what make the constraint 3*x + y >= 20

impossible to satisfy:

```
function model() {
    x <- bool();
    y <- int(0,100);
    z <- bool();
    t <- int(0,100);
    constraint 3*x + y >= 20;
    constraint pow(y,3) <= 250;
    constraint 4*z + t <= 18;
    maximize x*t + 8*z*y;
}
```

The computation of the inconsistency core can be launched in the output function as follows:

```
function output() {
    iis = computeInconsistency();
    println(iis);
}
```

The resulting output on the standard console is the following:

```
...
Run output...
Computing inconsistency core...
Inconsistency core found with 2 causes.
Irreductible inconsistency core found with 2 causes.
2 causes in inconsistency core:
        pow(int(0, 100)#1, 3)  <=  250
        3  *  bool()#0  +  int(0, 100)#1  >=  20
```

The two constraints responsible for the inconsistency are identified and displayed. Note that expressions are identified by their type and index. It is also possible to assign names to variables in the model function:

x.name = "x";
y.name = "y";
z.name = "z";
t.name = "t";

Having defined these names the inconsistency core reads as follows:

2 causes in inconsistency core:

pow(y, 3) <= 250

3 * x + y >= 20

□Note

In version 6.0, this functionality is offered in beta version: only explicit constraints are considered as potential causes of the inconsistency unless a single non-constraint expression is sufficient to cause the inconsistency. If the inconsistency core is empty it can mean that the problem is consistent or that its inconsistency derives of the combination of several implicit constraints.

6.3 At operator function

The operator “at” is an operator whose first operand is an array or a list and whose following operands are integers (the indices). The number of indices must match the dimension of the array (1 in case of a list). Its value is the value of the element in the array at the given indices.

Note
This page covers the use of the “at” operator on arrays (mono-dimensional or multi-dimensional). For a description of the use of the “at” operator on list variables, see List variables.

The 1-dimensional at operator

Assume that we model the behavior of a machine whose status can be on (1) or off (0). This status is modeled as a binary variable. Now the operating cost of this machine can depend on its status what would be modeled straightforwardly in LocalSolver as:

```
status <- bool();
mcost <- status ? onCost : offCost;
```

Note that onCost and offCost are not necessarily constants. They can depend on other expressions, for instance we may have an optional special technology (say a binary variable specialTechno) impacting the cost as follows: onCost <- 100 + 20 * specialTechno.

The at operator can be seen as a generalization of this conditional expression (c ? a : b). Let us generalize the above example to a machine having several possible operating modes. Now the status is an integer variable whose value is among {0, 1, 2, 3, 4, 5}, and the operating cost depends on the operating mode (the status).

Operating mode	Operating cost
0	Cost 0

Operating mode	Operating cost
1	Cost 1
2	Cost 2
3	Cost 3
4	Cost 4
5	Cost 5

Here again cost0, ... cost5 are not necessarily constants. They can depend on other expressions or decisions. It would be possible to model the operating cost of this machine with 5 nested conditional expressions but this approach would be tedious and inappropriate in case of hundreds of operating modes. Instead the at operator allows us to have all costs in an array and to define the operating cost by simply accessing this array with the index status:

```
status <- int(0,5);
costs = {cost0, cost1, cost2, cost3, cost4, cost5};

// In LSP the at operator is naturally written with the [] notation on a map
mcost <- costs[status];
```

In the APIs, the behavior is the same but requires a supplementary step. Indeed, although the LSP language enables you to directly use arrays, maps or lists, no such thing is possible for other languages and your array must be converted to an LSExpression before use. This LSExpression is an operator of type O_Array. It has no numeric value (calling getValue on it is forbidden) but contains an array of expressions instead. Expressions in the array can be constant or not.

To convert a C++ structure to a LocalSolver array, you can simply create an LSExpression with createExpression() then add each element manually as operands, or use the provided shortcut that takes iterators pointing to the beginning and the end of

your structure.

```
LSExpression status = model.intVar(0, 5);
std::vector<int> vv { cost0, cost1, cost2, cost3, cost4, cost5 };
// Convert the vector to LSExpression.
LSExpression lsv = model.array(vv.begin(), vv.end());
// Creates the at (operator [] is overloaded in the C++ API).
LSExpression mcost = lsv[status];
```

The multi-dimensional at operator

The at operator also provides a way to access arrays of arrays. It can be really convenient to avoid creating quadratic expressions.

Assume that we model a TSP where the ith city visited is stored in the expression city[i] and where the distance between each pair of cities is determined by the distance matrix distance. The distance travelled from the ith city to the next one can be simply defined by an at operator on the distance matrix with indices city[i] and city[i+1]:

```
city = {exprCity0, exprCity1, ..., exprCityn};
distance = {{d00, d01, ..., d0n}, {d10, d11, ...,d1n}, ..., {dn0, dn1, ..., dnn}};

// In LSP, even the multi-index at operator is naturally written
// with the [][] notation on a map
distanceTravelled <- distance[city[i]][city[i+1]];
```

As is the case with the 1-dimensional at, the APIs' equivalent of the above code also requires creating an O_Array expression, but this time it will be an array of arrays.

```
std::vector<LSExpression> city {exprCity0, exprCity1, ..., exprCityn};
std::vector<std::vector<int> > distance =
    {{d00, d01, ..., d0n}, {d10, d11, ...,d1n}, ..., {dn0, dn1, ..., dnn}};
LSExpression distanceArray = model.array();
for(int k = 0; k < n; k++)
    distanceArray.addOperand(model.array(distance[k].begin(), distance[k].end()));
LSExpression distanceTravelled = model.at(distanceArray, city[i], city[i+1]);
```

Jagged arrays

Arrays of arrays don't have to represent square matrices. They can perfectly store jagged arrays.:

```
a = { 1, 2, 3 };
b = { 4, 5 };
c = { 6 };

myArray <- {a, b, c};
```

Pitfalls

Arrays must have a uniform dimension

It is not required that arrays have the same number of elements (jagged arrays), but they must have the same dimension. Thus, it is not possible to mix, in the same expression, a 2-dimensional array, with a 3-dimensional array.

```
// Will throw an error because the first operand is an expression of
// dimension 0 (a boolean decision) whereas the second is an array of
// dimension 1
std::vector<int> a { 1, 2, 3 };
LSExpression array = model.array(a.begin(), a.end());
LSExpression error = model.array();
error.addOperand(model.bool());
error.addOperand(array);
```

Indices start at zero

The indexing of the array starts at 0. Hence when creating an at expression in the LSP language, the map must have a value defined for index 0.
For example, the following code will throw an exception:

```
status <- int(1,3);
costs[1] = 10;
cost[2] = 18.2;
cost[3] = 20;

// will throw an exception because indices of map "costs" do not start at zero
```

mcost <- costs[status];
Indeed when creating such a expression through a map, an O_Array expression is implicitly created, hence indices in the map must be consecutive and starting at 0, as in a regular array.

Implicit constraints are induced by the use of at operators

When the indices of an at expression take values outside of the bounds of the array, this expression is considered as violated and the LocalSolver solution has the “Infeasible” status (or “Inconsistent” if the solver can prove that no feasible solution exists).

For instance the following code has no explicit constraint but the model will be proven inconsistent because of the implicit constraints induced by the “at” operator:

```
x <- int(-3, 3);
c[i in 0..30] <- 3*i + 1;
minimize c[-3 + 2*x] + c[3 - 2*x];
```

In the APIs, an operator of type O_Array must be created

Altough the LSP language allows creating array expressions from maps, API functions require the explicit creation of an O_Array expression. See the C++ code below.

```
std::vector<int> vv { 10, 18, 20 };
LSExpression x = model.intVar(0, 2);

// Will not compile because vv is not an LSExpression
LSExpression wwx_bad = vv[x];

// OK since lsv is an LSExpression of type O_Array
LSExpression lsv = model.array(vv.begin(), vv.end());
LSExpression wwx_good = lsv[x];
```

6.4 Callback functions

LocalSolver allows you to react to specific events during the search by calling your own functions/procedures. It can be used for example to control when to stop the search or to display some specific information during the search.

□Note
This functionality requires using LocalSolver **through its APIs, it is not available in the LSP language.**

When adding a callback function to LocalSolver (with the addCallback() function) you can specify to which event your function must react, thanks to the LSCallbackType enumeration:
·PhaseEnded: your function will be called at the end of each search phase.
·PhaseStarted: your function will be called at the beginning of each search phase.
·Display: your function will be called periodically, after a display on the console or in the log file, every timeBetweenDisplays seconds.
·TimeTicked: your function will be called periodically, every timeBetweenTicks seconds.
·IterationTicked: your function will be called periodically, every iterationBetweenTicks iterations.

The same callback can be used for different events. The parameters timeBetweenDisplays, timeBetweenTicks and iterationBetweenTicks can be modified in the LSParam object.

When a callback is called, the solver is paused. In that state, you can call all the methods of the API marked as “allowed in state Paused”. For example, you can :
·Stop the resolution.
·Retrieve the current solution.
·Retrieve the statistics of the search.

In this section, we detail how callback functions are introduced in each programming language (Python, C++, Java, .NET). To illustrate this description we will use a simple example where a callback function allows to stop the search when no improvement of the objective function is found during a period of 5 seconds (the solved problem is a random knapsack).

In C++, a callback function is passed to LocalSolver as on object extending the localsolver::LSCallback class. This class has a single virtual method localsolver::LSCallback::callback() taking two parameters: the LocalSolver object that triggers the event and the type of the callback. It is possible to use the same callback object for multiple events or multiple LocalSolver instances.

Here we create a small class MyCallback containing two fields lastBestValue and lastBestRunningTime. The callback method uses these two fields with the statistics of LocalSolver to decide to stop or to continue the search. The callback is registered with the method localsolver::LocalSolver::addCallback():

```
class MyCallback : public LSCallback {
private:
    int lastBestRunningTime;
    lsint lastBestValue;
public:
    MyCallback() {
        lastBestRunningTime = 0;
        lastBestValue = 0;
    }

    void callback(LocalSolver& solver, LSCallbackType type) {
        LSStatistics stats = ls.getStatistics();
        LSExpression obj = ls.getModel().getObjective(0);

        if(obj.getValue() > lastBestValue) {
            lastBestRunningTime = stats.getRunningTime();
            lastBestValue = obj.getValue();
        }

        if(stats.getRunningTime() - lastBestRunningTime > 5) {
            cout << ">>>>>>> No improvement during 5 seconds: resolution is
stopped" << endl;
            ls.stop();
        } else {
            cout << ">>>>>> Objective improved by " << (obj.getValue() -
```

```
lastBestValue) << endl;
            }
        }
};

LocalSolver ls;
MyCallback cb;
ls.addCallback(CT_TimeTicked, &cb);
```

Each time this callback will be invoked by LocalSolver (namely every timeBetweenTicks seconds) it retrieves the statistics of the search and consider the total running time and the current best value of the objective function. If no improvement has been found during 5 consecutive seconds, it calls the stop() function to stop the search.

6.5 Black-Box Optimization

Black-Box Optimization¶

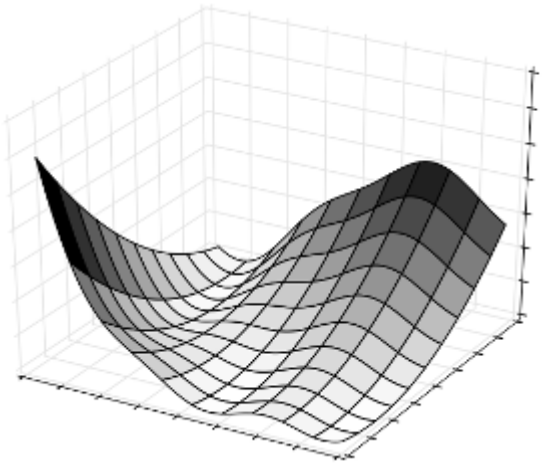
LocalSolver allows you to optimize function through a costly black-box interface. The solver is only able to evaluate the value of the function at a given point. This type of problems appears in practice when you need to simulate a physical process using an external library or simply when the mathematical description of the objective function is too complex to model.

Note

Note that this functionality requires using LocalSolver through its **LocalSolverBlackBox APIs, it is not available in the LSP language.**

In this section we detail how to optimize a black-box function in each programming language (C++, Java, .NET, Python). To illustrate this description we will minimize the branin function through a black-box interface. The number of evaluations will be limited to 20.

Branin function is defined by $f(x) = a(x2 - b*x1^2 + c*x1 - r)^2 + s(1-t)\cos(x1) + s$ with a=1, $b=5.1/(4\pi^2)$, $c=5/\pi$, s=10 and $t=1/(8\pi)$. The domains of x1 and x2 are respectively [-5,10] and [0,15].



For more details, see : branin.html

Black-Box optimization in C++¶

In C++, a black-box function is passed to LocalSolverBlackBox as an object extending the LSBBNativeFunction class. This class has a single virtual method call taking as parameter a LSBBNativeContext object. The``call`` method uses this context to access the decision values of the current point to evaluate and returns the value of the function at this point:

```
#include <iostream>
#include "api/blackbox/localsolverblackbox.h"

using namespace localsolverblackbox;

#define PI 3.14159

class Branin : public LSBBNativeFunction {
    virtual lsdouble call(const LSBBNativeContext& context){
        lsdouble x = context.getDoubleValue(0);
        lsdouble y = context.getDoubleValue(1);
        return pow(y - (5.1 / (4.0 * PI * PI)) * x * x + 5.0 / PI * x - 6, 2)
            + 10 * (1 - 1 / (8.0 * PI)) * cos(x) + 10;
    }
};
```

A LocalSolverBlackBox model is then created to optimize this black-box function. The black-box function is transformed into a LSBBExpression object using the createNativeFunction method. The decisions are associated to the function using a LSBBExpression object of type O_Call. The model is then parametrize to use 20 evaluations of the black-box function. The solution value can be obtained after the resolution using a LSBBSolution object:

```
int main(){
    Branin braninFunction;

    LocalSolverBlackBox ls;
    LSBBModel model = ls.getModel();

    LSBBExpression x = model.floatVar(-5,10);
```

```
    LSBBExpression y = model.floatVar(0,15);
    LSBBExpression f = model.createNativeFunction(&braninFunction);
    LSBBExpression call = model.call();
    call.addOperand(f);
    call.addOperand(x);
    call.addOperand(y);
    model.addObjective(call, OD_Minimize);
    model.close();
    ls.getParam().setEvaluationLimit(20);
    ls.solve();
    LSBBSolution sol = ls.getSolution();
    std::cout << "x=" << sol.getDoubleValue(x) << std::endl;
    std::cout << "y=" << sol.getDoubleValue(y) << std::endl;
    std::cout << "obj: " << sol.getDoubleValue(call) << std::endl;
}
```

Black-Box optimization in Java¶

In Java, a black-box function is passed to LocalSolverBlackBox as an object implementing the LSBBNativeFunction interface. This interface has a single method call taking as parameter a LSBBNativeContext object. The call method uses this context to access the decision values of the current point to evaluate and returns the value of the function at this point.
A LocalSolverBlackBox model is then created to optimize this black-box function. The black-box function is transformed into a LSBBExpression object using the createNativeFunction method. The decisions are associated to the function using a LSBBExpression object of type Call. The model is then parametrize to use 20 evaluations of the black-box function. The solution value can be obtained after the resolution using a LSBBSolution object:

```
import localsolverblackbox.*;

public class Branin{
    public static void main(String [] args) {

        LocalSolverBlackBox ls = new LocalSolverBlackBox();
        LSBBModel model = ls.getModel();
        LSBBExpression f = model.createNativeFunction(new LSBBNativeFunction(){
        public double call(LSBBNativeContext context){
            double x = context.getDoubleValue(0);
            double y = context.getDoubleValue(1);
            return Math.pow(y - (5.1 / (4.0 * Math.PI * Math.PI)) * x * x
                + 5.0 / Math.PI * x - 6, 2)
                + 10 * (1 - 1 / (8.0 * Math.PI)) * Math.cos(x) + 10;
        }
        });

        LSBBExpression x = model.floatVar(-5,10);
        LSBBExpression y = model.floatVar(0,15);
        LSBBExpression call = model.call();
        call.addOperand(f);
        call.addOperand(x);
```

```
            call.addOperand(y);
            model.addObjective(call, LSBBObjectiveDirection.Minimize);
            model.close();
            ls.getParam().setEvaluationLimit(20);
            ls.solve();
            LSBBSolution solution = ls.getSolution();
            System.out.println("x=" + solution.getDoubleValue(x));
            System.out.println("y=" + solution.getDoubleValue(y));
            System.out.println("obj:" + solution.getDoubleValue(call));
        }
}
```

Black-Box optimization in .NET¶

In .NET, a black-box function is passed to LocalSolverBlackBox as a delegate method taking as a single parameter a LSBBNativeContext object. The method uses this context to access the decision values of the current point to evaluate and returns the value of the function at this point. In the example we use a static method:

```
public static double BraninEval(LSBBNativeContext context) {
    double x = context.GetDoubleValue(0);
    double y = context.GetDoubleValue(1);

    return Math.Pow(y - (5.1 / (4.0 * Math.PI * Math.PI)) * x * x
        + 5.0 / Math.PI * x - 6, 2)
        + 10 * (1 - 1 / (8.0 * Math.PI)) * Math.Cos(x) + 10;
}
```

A LocalSolverBlackBox model is then created to optimize this black-box function. The black-box function is transformed into a LSBBExpression object using the CreateNativeFunction method. The decisions are associated to the function using a LSBBExpression object of type Call. The model is then parametrize to use 20 evaluations of the black-box function. The solution value can be obtained after the resolution using a LSBBSolution object:

```
public static void Main(string[] args)
{
    LocalSolverBlackBox ls = new LocalSolverBlackBox();
    LSBBModel model = ls.GetModel();
    LSBBExpression x = model.Float(-5,10);
    LSBBExpression y = model.Float(0,15);
    LSBBExpression f = model.CreateNativeFunction(BraninEval);
    LSBBExpression call = model.Call();
    call.AddOperand(f);
    call.AddOperand(x);
    call.AddOperand(y);
    model.AddObjective(call, LSBBObjectiveDirection.Minimize);
    model.Close();
    ls.GetParam().SetEvaluationLimit(20);
    ls.Solve();
```

```
        LSBBSolution sol = ls.GetSolution();
        Console.WriteLine("x="+sol.GetDoubleValue(x));
        Console.WriteLine("y="+sol.GetDoubleValue(y));
        Console.WriteLine("obj:"+sol.GetDoubleValue(call));
}
```

Black-Box optimization in Python¶

In Python, a black-function is simply a function or a method passed to LocalSolverBlackBox. This method only have a single parameter of type LSBBNativeContext object. The method uses this context to access the decision values of the current point to evaluate and returns the value of the function at this point:

```
def branin_eval(context):
    x = context.get(0)
    y = context.get(1)
    return math.pow(y - (5.1 / (4.0 * math.pi * math.pi)) * x * x
        + 5.0 / math.pi * x - 6, 2)
        + 10 * (1 - 1 / (8.0 * math.pi)) * math.cos(x) + 10
```

A LocalSolverBlackBox model is then created to optimize this black-box function. The black-box function is transformed into a LSBBExpression object using the create_native_function method. The decisions are associated to the function using a LSBBExpression object of type CALL. The model is then parametrize to use 20 evaluations of the black-box function. The solution value can be obtained after the resolution using a LSBBSolution object:

```
with localsolverblackbox.LocalSolverBlackBox() as ls:
    model = ls.get_model()
    x = model.float(-5,10)
    y = model.float(0,15)
    f = model.create_native_function(branin_eval)
    call = model.call()
    call.add_operand(f)
    call.add_operand(x)
    call.add_operand(y)
    model.add_objective(call, localsolverblackbox.LSBBObjectiveDirection.MINIMIZE)
    model.close()
    ls.get_param().set_evaluation_limit(20)
    ls.solve()
    sol = ls.get_solution()
    print "x=" + str(sol.get_value(x))
    print "y=" + str(sol.get_value(y))
    print "obj:" + str(sol.get_value(call))
```

6.6 List variables

In addition to boolean, integers and floats, LocalSolver offers a higher level decision variable: lists.

The list operator

The list operator allows defining a decision variable whose value is a collection of integers within a range `[0, n-1]` where n is the unique operand of this operator. Mathematically a list is a permutation of a subset of `[0, n-1]`: it does not necessarily contains all the values in `[0, n-1]` and all values in a list will be pairwise different, non negative and strictly smaller that n. Note that the unique operand of this operator must be a constant strictly positive integer.

For instance the following line creates a list decision variable of range 10:

```
x <- list(10);
```

The value of this list is obtained with the syntax `x.value` in the LSP language, and with the method `getCollectionValue()` in LocalSolver's APIs. It returns an object of type `LSCollection`, that can be read and modified through the methods: `count, get, clear, add`.

Modifying this `LSCollection` object modifies the value of the corresponding list variable. The code below illustrates the use of these methods:

```
println(x.value.count()); // Current size of the list

x.value.clear(); // empty the list

x.value.add(3); // add a value, throw an error if this value in not in
interval [0,9], if x was defined as list(10)

x.value.add(5); // add a value, throw an error if this value is already
included in the list

for[e in x.value] println(e); // print the content of the list

println(x.value); // print the content of the list, ``[3, 5]`` in this
case
```

Operators on lists

Unary and binary operators

The `count` operator returns the number of elements in a list. For example, the following model merely expresses the search for a list of maximum size:

```
x <- list(5);

maximize count(x);
```

The `at` operator allows accessing the value at a given position in the list. It takes two operands: a list and an integer expression (not necessarily constant). It returns -1 when the given index is negative or larger or equal to count(x).

For example, the objective function in the following model is to maximize the product of the first and last items in the list:

```
x <- list(5);

constraint count(x) > 0;

maximize x[0] * x[count(x)-1];
```

The `indexOf` operator returns the position of a given integer in the list or -1 if this integer is not included in the list. It takes two operands: a list and an integer expression (not necessarily constant). For example, given a matrix c of size n, the linear ordering problem consists in finding a permutation of [0..n-1] of minimum cost, where a cost c[i][j] is paid when j is before i in the ordering. Here is the corresponding model:

```
x <- list(n);

constraint count(x) == n;

minimize sum[i in 0..n-1][j in 0..n-1](c[i][j] * (indexOf(x,i) >
indexOf(x,j)));
```

N-ary operators

The `disjoint` operator applies to N lists sharing the same range. It takes value 1 when all lists are pairwise disjoint (that is to say that no value appears in more than one list), and value 0 otherwise. It takes at least one operand. In the following example we try to maximize the minimum size among three lists. Since they are constrained to be disjoint this maximum will be 3:

```
x <- list(10);

y <- list(10);

z <- list(10);

constraint disjoint(x, y, z);

maximize min(count(x), count(y), count(z));
```

The `partition` operator applies to N lists sharing the same range. It takes value 1 when the given lists form a partition of the set [0, n-1]. In other words, `partition(xlists)` is equivalent to `disjoint(xlists) && sum[i in 0..count(xlists)-1](count(xlists[i]) == n`. It takes at least one operand.

These operators are particularly useful when tasks have to be dispatched to different machines, or locations have to be dispatched to different trucks for instance.

Application to routing problems

In the context of routing problems, list variables can be used to model a variety of problems. A pure Traveling Salesman Problem (TSP) is modeled with a single list `x` with a constraint `count(x)==n` in order to specify that all cities must be visited. This constraint would be omitted for a Prize Collecting TSP, where a penalty is paid for cities not in the tour.

A vehicle routing problem (VRP) will be modeled with k lists if k is the number of trucks. For a classical VRP these lists will be constrained to form a partition (operator `partition`), whereas for a Prize-Collecting VRP only their disjointness will be required (operator `disjoint`).

Distances can be either given as a matrix and accessed with the `at` operator or explicitly computed (with operators `pow` and `srqt` for Euclidian distances for instance).

Detailed routing examples are available in *our example tour*.

6.7 Function operator(delegates)

LocalSolver offers the possibility to apply n-ary operators (like sum, min, max and so on) to a range of dynamic size. For instance it allows defining a sum whose number of terms will depend on other expressions. In such an expression, the iterated term is introduced as a function. A typical usage is for defining a sum over items of a list:

```
// with x a list
total <- sum(0..count(x)-1, i => quantity[x[i]]);
```

In the above example, `sum` has two operands:

- The first operand is a range of integers defined by its extremities.
- The second operand is a lambda expression defining the function that should be applied to each element in the range.

Both operands are LS expressions.

We will describe the properties of a range and how a function can be introduced. Then we will detail which operators can benefit from this feature, including the special case of the array operator. Several examples of our *our example tour* illustrate this feature (including the classical *CVRP*).

Ranges

A range is an collection of consecutive integers.

Warning

In the LSP modeling language, the .. operator defines a closed range that is to say that both extremities are included in the interval: a..b refers to the sequence of integers from a to b. **On the contrary**, when calling LocalSolver from its APIs, range(a,b) refers to the sequence of integers from a to b-1.

Both extremities of the range can be non-constant expressions. Note that some n-ary operators (like min and max) will be considered as undefined when the range is empty, that is to say that the solution will remain infeasible while the range is empty. For instance, the following expression implicitly constrains `b` to be larger or equal to `a`:

```
minimize max(a..b, i => v[i+1] * z);
```

Functions

A function is a particular LocalSolver expression composed of two parts, `arguments => body`:

The arguments of the function, which are also LS expressions, automatically and implicitely created when you use the `arguments => body` construct.

The body of the function. The body is an LS expression that will be used to evaluate the result of the function. The body can be any LS expression composed of any operands and operators supported by LocalSolver.

Note that these functions are explicitly defined with a combination of LocalSolver operators and **should not** be confused with *Native functions*.

Applying a function to a range

Applying a function to a range is achieved with the syntax `op(range,function)` where:

- `op` is some n-ary operator among: `sum`, `prod`, `min`, `max`, `and`, `or`, `xor`, `array`.
- `range` is a range of integers
- `function` is a function with exactly one argument, except for the `array` operator which accepts one or two operands (see below).

The value of such a `op(range,function)` expression can be computed as follows. For each integer `i` within `range`, `function(i)` is evaluated; and all these numbers are agregated with the operator `op`. If we define `v <- sum(a..b, i => f(i))`, then `v` will be equal to the sum of all `f(i)` for `i` in interval `[a,b]`.

A typical usage of this feature can be found in our *TSP example*, where the sum of distances along the circuit is computed as follows:

```
obj <- sum(0..nbCities-2, i => distanceWeight[cities[i]][cities[i+1]])
      + distanceWeight[cities[nbCities-1]][cities[0]];
```

Special case

When the `array` operator is used in this context, it creates an array whose size will vary with the size of the associated range. We allow a recursive definition of elements of this array by using a second argument in the function, containing the evaluation of the function on the previous element of the range.

Formally, if we define `v <- array(a..b, (i,prev) => f(i,prev))`, we will have `v[i] = f(i,v[i-1])` for all `i` in interval `[a,b]`, with `v[-1]` equal to 0 by convention.

The use of this feature can be illustrated on a routing problem with time-windows. Having opening hours on each location visited by a truck, we have to take into account the possible waiting time of the truck in case or early arrival. In fact the resulting time will be the maximum between the earliest arrival time (based on the driving time from the previous location) and the opening hour. Taking into account a service time on each location we have:

```
function departureTime(route, i, prev) {
   arrivalTime <- (i==0) ?
       openingHour[route[i]] :
       max(openingHour[route[i]], prev +
distance(route[i-1],route[i]));
   return arrivalTime + serviceTime[route[i]];
}
```

And the array of all departure times can be defined recursively as:

```
times <- array(0..count(route)-1, (i, prev) => departureTime(route, i,
prev));
```

Althoug we have used a function of the modeling language to obtain a more readable model it is important to note that `departureTime` merely returns an expression made of LocalSolver operators.

Now what if we also have closing hours at each location that is to say that the truck must have left location `L` before `closingHour[L]`? Such a constraint can be added applying operator `and` to the same range:

```
constraint and(0..count(route)-1, i => times[i] <= closingHou
```

6.8 Piecewise operator

Piecewise linear function are introduced in LocalSolver with the piecewise operator. A piecewise linear function is a function composed of straight-line sections. The extremities of each straight-line are called breaking points. We define a piecewise function by giving all its breaking points in left to right order.

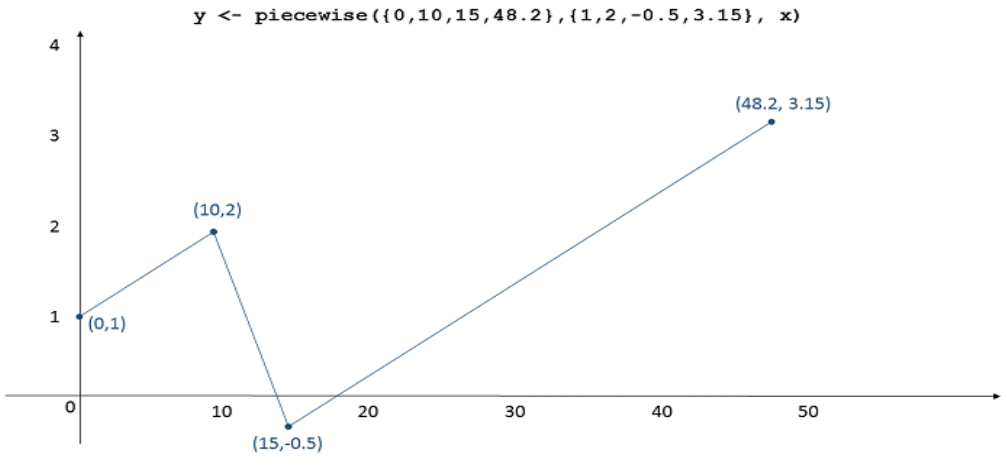
The piecewise operator takes exactly 3 arguments:

- A non-decreasing array of “n” constant numbers, with “n” >= 2.
- A array of “n” constant numbers
- An integer or double expression.

The solution will be infeasible if the value of the third operand is strictly smaller that the first element of the first array, or strictly larger than the last element of the first array. This operator returns a floating number.

The expression piecewise(x,y,z) returns the image of z by the function defined by geometric points (x[0], y[0]), (x[1], y[1]), ... (x[n-1], y[n-1]). For instance piecewise({0, 50, 100}, {0, 10, 100}, 75) returns 55.

The figure below illustrates the curve defined by y <- piecewise({0,10,15,48.2},{1,2,-0.5,3.15}, x). Note that this expression implicitly set constraints x >= 0 and x <= 48.2.



Discontinuities are allowed in the definition of the function, that is to say that two geometric points can share the same x-coordinate. By convention the value taken by the function at such a discontinuous point is the one associated to the last occurrence of this x-coordinate in array.

For instance piecewise({0, 50, 50, 100}, {0, 0.1, 0.9, 1}, 50) returns 0.9.

The figure below illustrates the curve defined by y <- piecewise({0 ,10 ,10 ,30 ,30 , 48.2}, {1 , 1, 2, 2, 3.15, 3.15}, x). This stepwise shape is often useful but any other discontinuous linear functions are allowed.