A guide for pit optimization with Pseudoflow in python

Sebastian Avalos (sebastian.avalos@queensu.ca)
Julian M. Ortiz (julian.ortiz@queensu.ca)

Abstract

The Pseudoflow algorithm is used to outline the ultimate pit limit by finding the maximum net value of the blocks extracted, while respecting precedences for their extraction. It works by modelling the blocks as a direct graph network and solving the minimum cut problem. Outlining the ultimate pit limit allows mine practitioners to design and plan an open pit operation. This guide provides a framework for running Pseudoflow in python along with a brief description of the method and a copper deposit application to exemplify its use.

1. Preliminary words

The aim of this guide is to provide a python framework for running the Hochbaum’s Pseudoflow (Hochbaum, 2008) algorithm in a merely research context (not-for-profit). The codes provided belong to a broader project in which block models built using the GSLib (Deutsch et al., 1992) codes are processed and analyzed by several python sub-routines. Thus, GSLib practitioners will find common grounds between the structure of GSLib output models and the block model representation matrix used herein.

We discuss the process to develop a direct graph representation network of a three-dimensional block model and how to impose geometrical precedence constraints, but we do not delve into the details of the Pseudoflow method or any of its variants. We rely on the non-commercial solver developed by Hochbaum (2008) and implemented in the PSEUDOFLOW python package released in July/2019 and updated in July/2020.

2. Ultimate pit limit

In open pit mining, the ultimate pit limit (UPL) defines the total rock to be mined across the life of the mine. The limits are outlined by solving an UPL algorithm under economic and spatial constraints along with a specific function to be maximized (minimized). Usually, the aim is to maximize the final net present value of the entire mine project, however, the UPL problem is defined based on the undiscounted value, that is, it maximizes the net value of the extracted blocks.

Conventionally, the ultimate pit limit problem has been solved by the Lerchs and Grossman (Lerchs, 1965) algorithm in two and three dimensions. The former uses a simple dynamic programming approach, while the latter relies on graph theory solving a maximum closure problem. Alternatively, by using a network flow representation, a minimum cut algorithm can be used to solve the ultimate pit limit problem (Hochbaum, 1998). The pseudoflow (Hochbaum, 1998, 2008) algorithm and its variations implement the minimum cut algorithm. These methods are characterized by their efficiency (Hochbaum, 2001) in contrast with conventional approaches and other maximum flow algorithms such as Ford-Fulkerson Algorithm (Ford and Fulkerson, 1956).

1 Cite as: Avalos, S., Ortiz, JM (2020) A guide for pit optimization with Pseudoflow in python, Predictive Geometallurgy and Geostatistics Lab, Queen’s University, Annual Report 2020, paper 2020-11, 186-194.
3. Pseudoflow representation

Let $G = (V, A)$ be a direct graph network with $n = |V|$ number of nodes and $m = |A|$ number of arcs. Let $(u, v) \in A$ be a direct arc from $u$ to $v$, then $c_{uv}$ and $f_{uv}$ represents the arc capacity (non-negative real number) and arc flow, respectively. By setting a lower bound capacity to zero, a pseudoflow $f$ assigns to each arc $(u, v)$ a flow $f_{uv}$ so that $0 \leq f_{uv} \leq c_{uv}$.

An s,t-graph $G_{st} = (V_{st}, A_{st})$ corresponds to an extension of $G$ with two additional nodes: a source $s$ and a sink $t$, $V_{st} = V \cup \{s, t\}$. The set of arcs $A_{st}$ now includes source-to-node arcs $A(s)$ and node-to-sink arcs $A(t)$, $A_{st} = A \cup A(s) \cup A(t)$. A closure graph is an s,t-graph whose arcs with finite capacities are only the arcs adjacent to the sources and sink nodes.

Several other terms and definitions, such as rooted tree, normalized tree, residual graph, inflow($u$), outflow($u$) and excess, are required to provide the step-by-step operation of the generic pseudoflow algorithm that maximizes the flow $f$. For interested readers we recommend Hochbaum (2008).

For our scope, solving the ultimate pit limit using the pseudoflow algorithm requires representing the ore body block model as a closure graph. The node set $V$ corresponds to all blocks in the block model. We call the $A(s) \cup A(t)$ arcs as external arcs and $A$ arcs as internal arcs. We assume that the block model has been economically evaluated (Figure 1 left) thus having negatives and non-negatives values representing the block valuations. All blocks with negative values are connected to the sink (Figure 1 center-top) and have a capacity equal to the absolute value of their economic value. The source is connected to all blocks with non-negative economic value (Figure 1 center-bottom) and their capacities are equal to the block economic value. This set of finite capacities corresponds to the external arcs. The internal arcs represent the geometrical precedence constraint (Figure 1 right) and their capacities are set to infinity. By having this representation, the pseudoflow algorithm can be applied to find the maximum flow going from the source to the sink considering the geometrical precedence. The maximum flow in the graph represents the maximum net value in the ore body, so the set of nodes involved in the maximum flow solution corresponds to the blocks in the resulting ultimate pit limit.

Figure 1: Example of arc creation in a 2D block model. Economic block value (left), external arcs source-node (center-bottom), external arcs node-sink (center-top), and internal arcs with 3 top block precedence (right).

It is worth mentioning that the previous s,t-graph is prone to find an ultimate pit limit with a “vertical wall” several benches high on the top of the pit, as the blocks in the boundary are not constrained by a precedence of three blocks, but only by two. In practice, this can be easily avoided by considering only the blocks inside the largest feasible pit. In other words, blocks that fall outside the largest feasible pit, should be excluded from the initial graph. This is the case for blocks in the bottom left and bottom right section.
of our example. Similarly, if different bench heights or slope angles are required, these must be imposed in the internal arcs, thus changing the geometrical precedence.

The following code implements both the previous idea of making a network over a maximum feasible pit and considering an static geometrical precedence. Additionally, the solver by Hochbaum (2008) implements a pseudoflow variant in which the capacity of nodes in \( A(s) \) is monotone non-decreasing using a factor \( \lambda \) and the capacity of nodes in \( A(t) \) is monotone non-increasing with the same \( \lambda \): \( \text{capacity} = \text{constant} + \text{multiplier} \cdot \lambda \). For our application, the constant is the capacity (i.e., the absolute block economic value), the multiplier is set to one, and \( \lambda \) is set to zero.

4. Python codes

The following codes run the 3D pseudoflow algorithm over a predefined block model. A function-by-function presentation is used along with comments and descriptions. We begin by importing four python packages (or libraries):

1. `import numpy as np`
2. `import networkx as NetX`
3. `import pseudoflow as pf`
4. `import time`

**NumPy** (Walt et al., 2011) operates over large and multi-dimensional arrays and matrices; **NetworkX** provides the structures for networks and graph representation (nodes, arcs, capacities, etc.) (Hagberg et al., 2008); **PSEUDOFLow** (Hochbaum, 2008) provides a variant of the fully parametric implementation of Hochbaum’s Pseudoflow (HPF) algorithm for minimum cut on directed graphs; **TIME** provides several time-related functions.

```python
1 def Pseudoflow_UPL (BM, nx, ny, nz, VarIn, VarOut):
2     print("Beginning Pseudoflow")
3     start_UPL = time.time()
4     source = 0
5     sink = np.int(nx*ny*nz + 1)
6
7     # Graph creation
8     Graph = NetX.DiGraph()
9
10    # External arcs creation by external function. Source - Nodes , Nodes - Sink
11    Graph = CreateExternalArcs (BM, nx, ny, nz, Graph = Graph , Var = VarIn )
12
13    # Internal arcs creation by external function. Block precedence (1 x5 or 1x9)
14    for ind_z in range(nz - 1):
15        pos_z = nz - ind_z - 2
16        for pos_y in range(ind_z + 1, ny - ind_z - 1):
17            for pos_x in range(ind_z + 1, nx - ind_z - 1):
18                # Precedence of 5 blocks
19                Graph = CreateInternalArcs1x5 (pos_x, pos_y, pos_z, nx, ny, Graph = Graph )
20                # Precedence of 9 blocks
21                Graph = CreateInternalArcs1x9 (pos_x, pos_y, pos_z, nx, ny, Graph = Graph )
22
23    # Solving the minimum cut problem via pf.hpf solver
24    RangeLambda = [0]
25    breakpoints, cuts, info = pf.hpf (Graph, source, sink, const_cap="const", mult_cap="mult ", lambdaRange=RangeLambda, roundNegativeCapacity=False)
26
27    # Going over the cuts.items finding the nodes inside the resulting UPL.
28    B = {x:y for x, y in cuts.items() if y == [1] and x!=0}
29    InsideList = list(B.keys())

2https://pypi.org/project/pseudoflow/
3https://docs.python.org/3/library/time.html
```

© Predictive Geometallurgy and Geostatistics Lab, Queen’s University 188
# Set all blocks as zero
BM[:, VarOut] = 0

for indUPL in range(len(InsideList)):
    # Set blocks inside UPL as one
    BM[np.int(InsideList[indUPL] -1),VarOut] = 1

print("--> Pseudoflow time: --%s seconds " % (np.around((time.time() - start_UPL), decimals=2)))

return BM

Listing 1: Pseudoflow function. Graph creation. Minimum cut solver. UPL block model flagging.

The above function Pseudoflow_UPL(BM, VarIn, VarOut) receives as input the block model (BM), the economic value column (VarIn) and the resulting UPL column (VarOut). Although the above python listing includes comments, we review the function step-by-step. In network terms, every block must be transformed into a node. Two additional nodes are required: Source node (coded as 0) and Sink node (coded as nx·ny·nz+1) where nx, ny, nz corresponds to the total nodes in the x, y and z axes, respectively.

An empty direct graph is created in line 8. Arcs capacities must be added to the graph. External arcs are created by calling the CreateExternalArcs() function. Internal arcs are created by calling the CreateInternalArcs1x5() or CreateInternalArcs1x9() function inside a nested for loops. We describe these functions in the below code listings. The nested for loop covers all blocks with geometrical precedence.

Once the direct graph is fully informed, it is time to solve the minimum cut problem. The parametric variant of the HPF algorithm implemented in the solver requires to set lambda values so the capacity of source-adjacent arcs is monotone non-decreasing in lambda while the capacity of sink-adjacent arcs is monotone non-increasing in lambda. We set the lambda range to just one value (zero) without loss of generality, so the capacities remain constant throughout the process. For further details, please refer to Hochbaum (2008). The solver is called by pf.hpf(Graph, source, sink, const_cap="const", mult_cap="mult", lambdaRange =RangeLambda, roundNegativeCapacity=False) where Graph is our created direct graph, source and sink were previously defined, const_cap is set to constant (arcs attribute with the constant capacity), mult_cap is set to mult (arcs attribute with the lambda multiplier), lambdaRange is associated with the RangeLambda vector, and negative capacities are not rounded to zero.

Line 28 goes over the cuts items, selecting the nodes inside the resulting UPL while line 29 converts the dictionary keys to a vector. Line 32 sets all blocks as zero in the corresponding block model column while line 36 assigns a value of one to all blocks inside the resulting ultimate pit limit. Finally, the function prints the time it took to generate the UPL and returns the updated block model.

def CreateExternalArcs(BM, nx, ny, nz, Graph, Var):
    Sink = np.int(nx*ny*nz + 1)
    for t_z in range(nz):
        pos_z = nz - t_z - 1
        for t_y in range(t_z, ny-t_z):
            for t_x in range(t_z, nx-t_z):
                p_i = 1 + t_x + nx*t_y + ny*nx*pos_z
                Capacity = np.absolute(np.around(BM[p_i-1,Var], decimals=2))
                if BM[p_i-1,Var] < 0:  #Negative local Economic Value
                    Graph.add_edge(p_i, Sink, const=Capacity, mult=1)
                else:
                    Graph.add_edge(0, p_i, const=Capacity, mult=1)
    return Graph

Listing 2: Create external arcs function.

The CreateExternalArcs(BM, nx, ny, nz, Graph, Var) function receives the block model (BM), the number of blocks on each x, y and z axes (nx, ny, nz), the column number of economic value (Var), and the working graph (Graph). Note how the for loops begin from the z-axis, then the y-axis and end with the x-axis, so the node number in the graph (pi value) of the block associated with the ith row in the block
model is defined as $1 + t_x + nx \cdot t_y + ny \cdot nx \cdot pos_z$ (see Table 1). The latter is a simple notation to move back and forth between the $[x,y,z]$ coordinates in the block model and the $[p_i]$ node position in the graph.

The following step is to connect each node (each block) with either the source node or the sink node. The nodes whose corresponding blocks have negative economic values are linked to the sink while nodes with a non-negative economic blocks associated are linked to the source. The resulting arcs, either source-node or node-sink, have a capacity equal to the absolute magnitude of the block economic value. We have defined that all external arcs have a multiplier equal to 1.

<table>
<thead>
<tr>
<th>Node number ($p_i$)</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$p_i$</td>
<td>$t_x$</td>
<td>$t_y$</td>
<td>$pos_z$</td>
</tr>
<tr>
<td>$nx \cdot ny \cdot nz$</td>
<td>$nx - 1$</td>
<td>$ny - 1$</td>
<td>$nz - 1$</td>
</tr>
</tbody>
</table>

Table 1: Node number in graph ($p_i$) and the corresponding position in the block model.

```python
1  def CreateInternalArcs1x9(pos_x, pos_y, pos_z, nx, ny, Graph):
2      p_0 = 1 + pos_x + nx*pos_y + ny*nx*pos_z
3
4      p_1 = 1 + (pos_x-1) + nx*(pos_y-1) + ny*nx*(pos_z+1)
5      p_2 = 1 + (pos_x) + nx*(pos_y-1) + ny*nx*(pos_z+1)
6      p_3 = 1 + (pos_x+1) + nx*(pos_y-1) + ny*nx*(pos_z+1)
7      p_4 = 1 + (pos_x-1) + nx*(pos_y) + ny*nx*(pos_z+1)
8      p_5 = 1 + (pos_x) + nx*(pos_y) + ny*nx*(pos_z+1)
9      p_6 = 1 + (pos_x+1) + nx*(pos_y) + ny*nx*(pos_z+1)
10     p_7 = 1 + (pos_x-1) + nx*(pos_y+1) + ny*nx*(pos_z+1)
11     p_8 = 1 + (pos_x) + nx*(pos_y+1) + ny*nx*(pos_z+1)
12     p_9 = 1 + (pos_x+1) + nx*(pos_y+1) + ny*nx*(pos_z+1)
13
14      Graph.add_edge(p_0, p_1, const=99e99, mult=1)
15      Graph.add_edge(p_0, p_2, const=99e99, mult=1)
16      Graph.add_edge(p_0, p_3, const=99e99, mult=1)
17      Graph.add_edge(p_0, p_4, const=99e99, mult=1)
18      Graph.add_edge(p_0, p_5, const=99e99, mult=1)
19      Graph.add_edge(p_0, p_6, const=99e99, mult=1)
20      Graph.add_edge(p_0, p_7, const=99e99, mult=1)
21      Graph.add_edge(p_0, p_8, const=99e99, mult=1)
22      Graph.add_edge(p_0, p_9, const=99e99, mult=1)
23
24      return Graph
```

Listing 3: Create internal arcs function. Nine blocks precedence.

```python
1  def CreateInternalArcs1x5(pos_x, pos_y, pos_z, nx, ny, Graph):
2      p_0 = 1 + pos_x + nx*pos_y + ny*nx*pos_z
3
4      p_2 = 1 + (pos_x) + nx*(pos_y-1) + ny*nx*(pos_z+1)
5      p_4 = 1 + (pos_x-1) + nx*(pos_y) + ny*nx*(pos_z+1)
6      p_5 = 1 + (pos_x) + nx*(pos_y) + ny*nx*(pos_z+1)
7      p_6 = 1 + (pos_x+1) + nx*(pos_y) + ny*nx*(pos_z+1)
8      p_8 = 1 + (pos_x) + nx*(pos_y+1) + ny*nx*(pos_z+1)
9      p_9 = 1 + (pos_x+1) + nx*(pos_y+1) + ny*nx*(pos_z+1)
10
11      Graph.add_edge(p_0, p_2, const=99e99, mult=1)
12      Graph.add_edge(p_0, p_4, const=99e99, mult=1)
13      Graph.add_edge(p_0, p_5, const=99e99, mult=1)
14      Graph.add_edge(p_0, p_6, const=99e99, mult=1)
15      Graph.add_edge(p_0, p_8, const=99e99, mult=1)
16      Graph.add_edge(p_0, p_9, const=99e99, mult=1)
17
18      return Graph
```

Listing 4: Create internal arcs function. Five blocks precedence.
The functions \texttt{CreateInternalArcs1x9()} \texttt{CreateInternalArcs1x5()} receives the graph, the position of the block model and the number of blocks in x and y directions. The former establishes the node position of the main block and the nine node positions of the above nine blocks while the latter does the same but only with the five blocks above, according to the geometrical precedence scheme. Then both add the arcs that link the node of the main block with the nodes of the upper blocks with an infinite capacity and a multiplier of one. Lastly, the updated graph is returned.

```python
def main():
    print("Start")
    start_time = time.time()
    nx, xmn, xsiz = 44, 24300, 16
    ny, ymn, ysiz = 62, 24800, 16
    nz, zmn, zsiz = 26, 3600, 16
    BlockModel = np.loadtxt("BM_matrix.txt") # Import Block Model
    BlockModel = Pseudoflow_UPL(BM=BlockModel, nx=nx, ny=ny, nz=nz, VarIn=4, VarOut=5)

    '''Save Block Model'''
    np.savetxt(fname ="BM_matrix.txt", X=BlockModel, fmt='%.3 f', delimiter='	')

    return print("--%s seconds of the whole process --" % (np. around((time.time()-start_time), decimals=2)))

if __name__ == "__main__":
    main()
```


The \texttt{main()} function is quite simple. It sets the start time, the block model dimensions, it imports the block model in text format delimited by tab and then it calls for the pseudoflow function. Once the process is done, it saves an updated block model and prints the running time.

5. Application

We demonstrate the resulting UPLs on a porphyry copper deposit. A block model of $44 \times 62 \times 26$ blocks of size $16 \times 16 \times 16$ is considered using the copper grade as key interest element (Figure 2 left). The economic value of each block is obtained via $V = (P - C_{p}) \cdot R \cdot T \cdot \alpha \cdot Z - T \cdot C_{m} - T \cdot C_{g}$ where $\alpha \in \mathbb{R}^{+}$ is a unit conversion factor, and $C_{m}, C_{p}, P, C_{p}, R, Z$ are process specific referring to mining cost, crushing-grinding cost, ore price, processing cost, ore recovery, and copper grade respectively. The Table 2 summaries the assumed values of each parameter and Figure 2 (right) shows the resulting economic block model.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price ($P$)</td>
<td>1.8</td>
</tr>
<tr>
<td>Processing cost ($C_{p}$)</td>
<td>0.315 usd/lb</td>
</tr>
<tr>
<td>Recovery ($R$)</td>
<td>80 %</td>
</tr>
<tr>
<td>Tonnage ($T$)</td>
<td>2.7\cdot16,Ton</td>
</tr>
<tr>
<td>Conversion factor ($\alpha$)</td>
<td>2204 lb/ton</td>
</tr>
<tr>
<td>Mining cost ($C_{m}$)</td>
<td>4.0 usd/ton</td>
</tr>
<tr>
<td>Crushing-grinding cost ($C_{g}$)</td>
<td>10.0 usd/ton</td>
</tr>
</tbody>
</table>

Table 2: Parameters for block economic assessment

In Numpy array format, the block model (BM\_matrix.txt) has a shape of (70928, 6) where 70928 corresponds to the $44 \times 62 \times 26$ blocks and 6 corresponds to six columns, each one representing the coordinates x, y, z, and the values of copper grade, economic value, and ultimate pit limit, respectively.

By running the code, the \texttt{main()} function is executed. First, the block model dimensions are defined as Numpy array. The \texttt{Pseudoflow\_UPL()} function is executed with arguments: \texttt{BM=BlockModel, VarIn=4, VarOut=5}, representing the block model to be analyzed, the column of the economic attribute and the column for the resulting UPL, respectively.

© Predictive Geometallurgy and Geostatistics Lab, Queen’s University
When calling the `Pseudoflow_UPL()` function, the external arcs of the network are created by calling the sub-function `CreateExternalArcs()`. The internal arcs result by assuming geometrical precedence either a $1 \times 5$ or $1 \times 9$ meaning that it is required to extract the above 5 blocks in cross shape or the above square of 3 by 3 blocks, respectively. The former is performed by calling the sub-function `CreateInternalArcs1x5()` while the latter by the sub-function `CreateInternalArcs1x9()`. In practice, the code requires to comment/uncomment line 20 and 22 of the `Pseudoflow_UPL()` function.

Based on the economic block model, the Figure 3 illustrates the resulting ultimate pit limits generated by the $1 \times 5$ and $1 \times 9$ geometrical constraints at the top and bottom of the figure, respectively. The former UPL contains 7,313 blocks with a total net value of 322.7 MUSD while the latter has 5,141 blocks with a total net value of 284.9 MUSD.

We extend the example by developing a simple nested pit analysis using the $1 \times 5$ geometrical constraint. The revenue factor varies between [0.3, 1.2] using a discretization of 0.02. Figure 4 presents the total tonnage inside the resulting pit, the corresponding tonnage of ore and its associated copper grade. Based on total
tonnage increments, we have arbitrarily chosen six pits at revenue factors 0.7, 0.84, 0.92, 1.0, 1.08 and 1.2. The associated nested pits are shown in Figure 5.

Figure 4: Nested pits via revenue factor analysis.

Figure 5: Six nested pits flagged from the smallest (six) to the largest (one). Blocks not extracted are coded as zero.
In terms of computing time, each run of the code took 1.5 seconds on average using a single workstation with a 64-bit Windows 10 Home system, Intel(R) Core(TM) i7-8700 3.70 GHz, 16 GB of RAM, solid state hard drive and an NVIDIA GeForce GTX 1080 Ti 11 GB.

6. Final comments

The previous code runs end-to-end. It has been developed for research purpose only. We encourage further implementations on specific geometrical precedence and different slop angles. The codes for economic assessment and visualization are not provided. Readers are welcome to use the code and if any bug is found please contact the author via email.

7. Acknowledgements

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), funding reference number RGPIN-2017-04200 and RGPAS-2017-507956.

8. Code availability

The code is available at the Geomet group repository: https://github.com/geometatqueens/2020-Pseudoflow-Python

9. Bibliography