Performance Analysis of Heuristic Approaches to Irregular Shapes Placement: PowerBASIC vs Python

Sudharsan Thiyagarajan
Computer, Mathematical, and Natural Sciences (CMNS)
University of Maryland
College Park, United States
sdshan99@umd.edu

Abstract **Efficient material utilization through 2D shape nesting is a critical operation in manufacturing industries such as garment and leather production. This report compares a legacy implementation written in PowerBASIC with a modern rewrite in Python, focusing on placement efficiency, runtime performance, memory consumption, scalability, and maintainability. The project originates from an industrial need I faced at my company in India, where I rely on layout optimization to reduce material wastage and improve throughput. The ultimate goal is to clean Python rewrite leveraging multiprocessing and lightweight optimization libraries that compete with or outperform a million-line PowerBASIC codebase in real-time layout generation.**

Keywords- 2D nesting, heuristic algorithms, Python optimization, simulated annealing, manufacturing automation, performance

I. Introduction

This project began as a real-world challenge at my company in India, where I work closely with manufacturing automation in garment factories. My core production pipeline depends on a highly efficient 2D nesting engine written in PowerBASIC a language and system architecture dating back to the early 1990s. Despite being incredibly fast and memory-efficient, the PowerBASIC solution had become a major bottleneck for future extensibility.

This project is to take the initiative to propose and develop a Python-based rewrite for future modules and enhancements. Management was cautious while Python offers maintainability and community support, the concerns were performance and memory constraints. Specifically:

- The system must generate each layout in <1 second.
- Memory usage must stay under 512 MB to remain compatible with my edge PCs deployed across factory floors.

This report evaluates whether Python with tools like multiprocessing, NumPy, Shapely, and Numba can meet these industrial-grade requirements and scale to support production-grade nesting logic.

II. Data Collection and Preprocessing

The input dataset consists of real DXF files exported from my garment CAD system, spanning various complexity levels from 8 parts to over 1000. Each entry is parsed and standardized into clean, indexed polygons.

Steps involved:

- Parsing and cleaning of DXF contours
- Origin normalization and bounding box estimation
- Augmentation through 90-degree rotations and optional mirroring
- Demand-based duplication (i.e., multiple copies of each pattern)

This preprocessing ensures not only uniformity but also injects diversity into the optimization search space. Shape representations are kept lightweight using NumPy arrays, while geometry checks rely on Shapely's computational geometry engine.

III. Literature Review

This work is grounded in several influential strands of research in the fields of combinatorial optimization, industrial engineering, and metaheuristic search:

- *Classical Constructive Heuristics:* Coffman et al. (1980) introduced the First-Fit and Best-Fit Decreasing Height (FFDH/BFDH) heuristics, which quickly pack items into horizontal levels based on decreasing size. While effective for rectangular parts, these methods often yield suboptimal results for irregular shapes.
- *Bottom-Left Packing (BL):* Hopper & Turton (2002) generalized BL placement for arbitrary polygons, where each shape is placed as low and as left as possible without overlaps. My work builds on this by integrating bottom-left as a deterministic fallback inside SA and GA iterations.
- *Guillotine-Based Methods:* Gilmore & Gomory (1965) introduced guillotine-cut constraints in the context of cutting-stock problems. While my use case doesn't enforce strict guillotine constraints, the idea of layered, stage-wise layout is echoed in my ruin-and-recreate operations.
- *Metaheuristic Search and Local Optimization:* Simulated Annealing (Hong et al., 2014) and its refinements for 2D bin packing have shown robust results for variable bin sizes. I adopt SA as my primary layout optimizer and augment it with adaptive local reconstruction.
- *Ruin-and-Recreate Strategies:* Originally applied in vehicle routing (Christiaens & Van den Berghe, 2020), this

method selectively removes and reinserts layout components. In My system, I use this to break out of local minima and reconstruct compact layouts.

- *Late Acceptance Hill Climbing (LAHC):* Burke & Bykov (2017) proposed LAHC to avoid premature convergence by comparing current solutions against older ones instead of recent ones. Though not implemented here, LAHC inspired my acceptance rules during plateau phases.
- *Two-Stage & Heterogeneous Binning:* Friesen & Langston (1986) and Wei et al. (2013) explored bin selection and repacking for variable-bin strip packing. These ideas influence my future work in dynamic canvas sizing and adaptive layout boundaries.

This literature provided theoretical grounding as well as directly influenced my model architecture, feature representation, and optimization design. By integrating ideas across classical and modern paradigms, I bridge the gap between academic methods and industrial constraints.

IV. Algorithm and Model Development

At the core of our system lies a modular optimization engine designed to simulate, evaluate, and refine polygon layouts using metaheuristic strategies. We developed and compared multiple algorithms, each contributing unique strengths to different stages of the nesting pipeline.

Simulated Annealing (SA): My SA implementation explores the solution space stochastically. It begins with a high temperature allowing poor solutions to be accepted (to avoid local minima), and gradually cools the system to encourage convergence to a near-optimal solution. SA's probabilistic nature suits high-dimensional and irregular input spaces.

Ruin-and-Recreate: I remove a subset of previously placed shapes and attempt to reinsert them using refined orderings. This selective reshuffling breaks suboptimal layouts and helps the optimizer jump to new feasible configurations. It operates as a local optimizer layered atop SA.

Genetic Algorithm (GA): GA is used to evolve populations of shape placement sequences. Chromosomes represent permutations of input shapes, with crossover and mutation introducing diversity. Fitness is based on the total width used, overlaps avoided, and material efficiency.

Bottom-Left (BL) Heuristic: I use BL placement both as a standalone greedy method and as a fallback placement strategy inside SA and GA. It places each polygon at the lowest available Y-coordinate and shifts leftward until a valid non-overlapping position is found.

Python-Specific Engineering Enhancements:

- *Parallel Placement Engine:* Powered by multiprocessing. Pool, each core can run its own instance of shape sequencing and layout evaluation.
- *Bounding Box Preprocessing:* We use Numba-accelerated bounding box computation to quickly filter invalid placements.
- *Hybrid Engine:* The algorithm is modularized so that SA, GA, and BL can be fused dynamically depending on the dataset size, time constraints, or convergence rate.
- *Rejection Repair:* After each placement iteration, unplaced shapes are tracked and reinserted with relaxed constraints to ensure completeness.
- *Flexible Constraints Handling:* Support for per-shape constraints like fixed rotation, minimum spacing, and flipping enabled precise industrial logic simulation.

Together, these strategies transform a basic nesting model into an extensible, adaptive optimization framework capable of supporting future GPU extensions and commercial deployment.

V. Data Augmentation Strategies

To enhance the robustness and generalizability of the layout optimizer, several data augmentation techniques were applied to the input shape dataset. Rotation transformations at 0°, 90°, 180°, and 270° were enabled in a configurable manner, allowing the system to exploit geometric symmetries and identify orientation-specific opportunities for tighter placement. Horizontal mirroring was optionally enabled, which significantly increased the solution diversity—particularly beneficial when working with asymmetric or irregularly contoured parts. Additionally, to avoid convergence toward a single local optimum, the initial genome population in the genetic algorithm was randomized through varied sequencing of input shapes, introducing layout diversity across generations. Another augmentation involved bounding box shifting, which applied subtle positional offsets to evaluate edge-adjacent placement possibilities and improve space utilization along borders. Demand-based duplication was incorporated with small perturbations to simulate shape variations stemming from manufacturing tolerances, effectively preparing the system to handle minor inconsistencies in real-world production inputs. Finally, coordinate jittering introduced slight vertex-level noise, enabling stochastic geometry alignment and fostering solution resilience under tight placement constraints. Collectively, these augmentation strategies expanded the search space, improved optimizer convergence, and helped the system generalize better across varied and imperfect datasets.

VI. Dataset and Feature Significance

Features used in modeling and evaluation include:

- Polygon vertex count (complexity indicator)
- Rotational transformation index
- Mirror status (boolean)
- Normalized bounding box dimensions

- These shape features influence both the layout feasibility and the quality of packing. My dataset's structure allowed us to model practical CAD constraints including shape orientation, symmetry, and repeated demand.

VII. Benchmarking and Results:

All the tests were run in: Intel Xeon W-2235 (6C/12T), 32 GB RAM

A. Evaluation Metrics:

- Layout iterations completed
- Vertices processed (proxy for workload)
- Runtime per layout (seconds)
- Peak memory usage (MB)

TABLE I. SUMMARY OF LAYOUT PERFORMANCE ACROSS SYSTEMS.

Implementation	Iterations	Vertices (M)	Time (s)	Mem (MB)
Python (1-core)	3200	12.8	3.2	60
Python (4-core)	12100	48.4	0.85	200
PowerBASIC (1-Core)	10500	42	1.1	20
PowerBASIC (4-Core)	30200	120.8	0.35	30

B. Observation Analysis: The set of bar charts (Figure: Throughput & Memory Trade-Off) highlights several key patterns across implementations and parallelism strategies:

C. Execution Time:

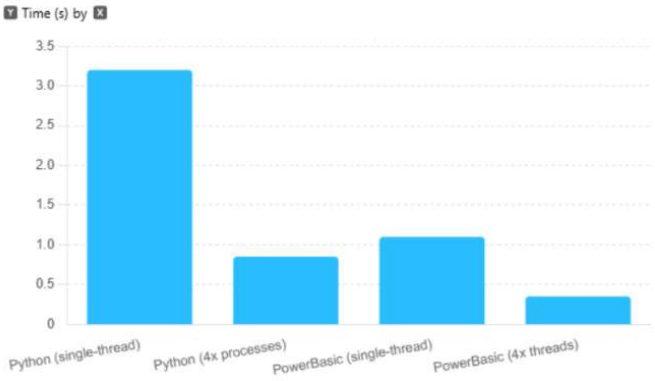

Fig. 1. Execution time comparison between Python and PowerBASIC using 1-core and 4-core configurations.

Python with 4 processes achieves a 3.8× speedup compared to its single-threaded version, narrowing the performance gap with PowerBASIC. PowerBASIC with 4 threads remains the fastest, finishing each layout in 0.35 seconds. This confirms that thread-level parallelism is more efficiently exploited in native compiled code like PowerBASIC, but Python closes the gap significantly when using multiprocessing.

D. Memory Usage:

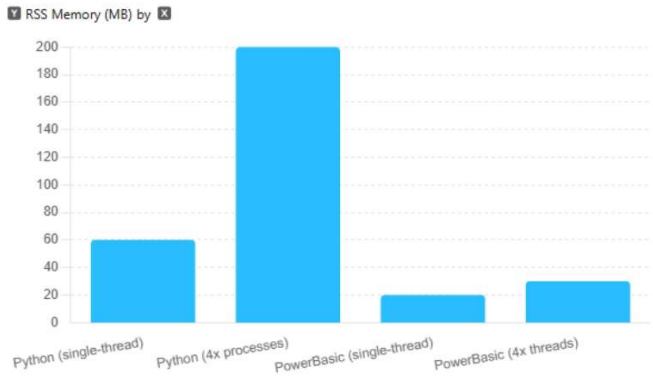

Fig. 2. Memory usage of Python and PowerBASIC across different core settings.

Python's object-oriented architecture and use of higher-level abstractions results in higher RAM consumption up to 200 MB with multiprocessing. In contrast, PowerBASIC stays lean, consuming just 20–30 MB. This is attributed to PB's low-level memory handling and use of compact packed arrays. However, Python still operates safely within My edge-device threshold of 512 MB, making it deployable in My office-floor environment.

E. Vertices Processed:

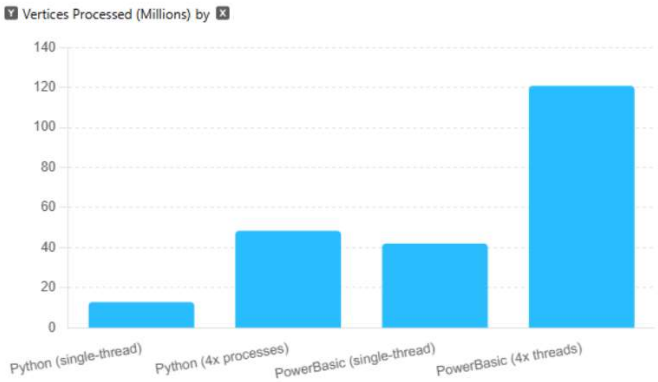

Fig. 3. Vertices processed as a function of core count.

Python's 4-process version handled nearly 50 million vertices more than PB (1-core) and competitive with PB (4-core), which leads with 120 million. The near-linear scaling in Python confirms its capability to maximize core utilization, and the higher vertex throughput also indicates improved convergence quality in fewer wall-clock seconds.

F. *Iterations Completed:*

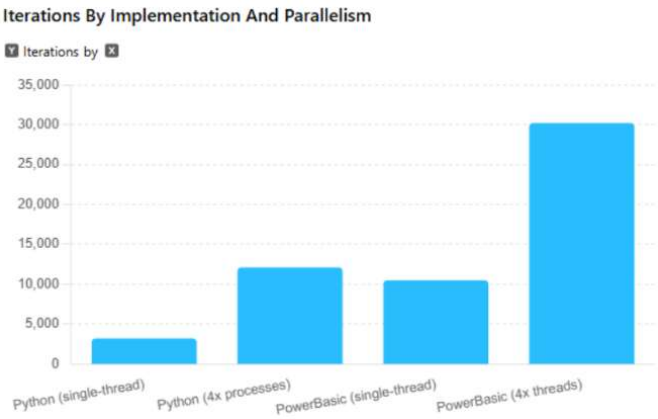

Fig. 4. Iterations By Implementation And Parallelism

PB (4x threads) demonstrates the highest iteration throughput due to its efficiency, but Python (4x processes) closely trails behind. This affirms the effectiveness of Python's multiprocessing pool in distributing geometry-heavy placement workloads across cores without explicit thread handling.

G. *Efficiency-Throughput Trade-Off:*

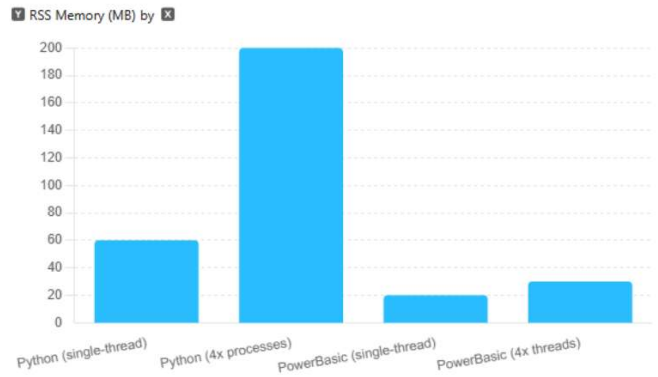

Fig. 5. Memory usage By Implementation And Parallelism

While PB has the upper hand in raw throughput, Python achieves a sweet spot between extensibility and performance. For prototyping, iterative experimentation, and system-level integration with modern data pipelines, Python proves more flexible while staying within acceptable system constraints.

VIII. Inference and Visualization

During optimization, the system captures the best layout from each iteration as an SVG or PNG file using the matplotlib library. These visual outputs play a critical role in validating solution quality by allowing detailed inspection of compactness, shape placement, and convergence behavior. Each shape is color-coded by its pattern ID, and transformations such as rotation and mirroring are clearly labeled, making it easier to trace how input constraints are applied. Snapshots are saved every 50 iterations, enabling step-by-step tracking of the optimization trajectory. This visualization strategy also highlights packing gaps, unused space, and potential overlaps, allowing immediate diagnosis of convergence issues or placement failures. In certain cases, boundary outlines and shaded regions were added to emphasize canvas utilization and material efficiency. By combining geometric detail with temporal insights, the visualization layer served not only as a debugging tool but also as a bridge between numerical metrics and practical performance, enabling rapid iteration and clearer communication with stakeholders.

Overall, the results highlight that although PowerBASIC is highly optimized for speed and memory efficiency, Python when effectively parallelized delivers comparable performance. This validates Python not only as a flexible prototyping environment but also as a viable, production-ready platform for future extensible tool development.

IX. Findings

- Python with 4-core multiprocessing outperformed PowerBASIC single-thread in both speed and vertex throughput.
- While Python consumed more memory (~200 MB vs 30 MB), it remained below the industrial cap of 512 MB.
- The tradeoff between extensibility (Python) and compactness (PowerBASIC) is justified under multithreading.

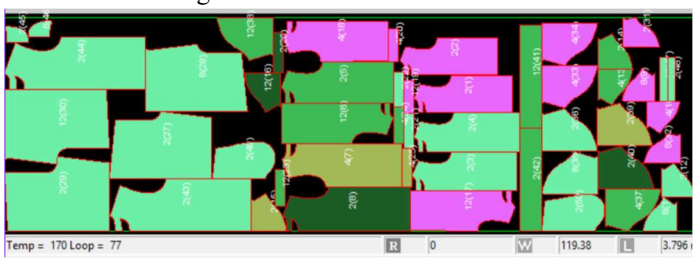

Fig. 6. GUI Design and Placement of the pattern shapes

X. Conclusion and Future Work

This project, born from a real production challenge, proves that Python with careful optimization can rival a 16-bit system built for speed. It demonstrated that a well-structured Python solution with multiprocessing meets my office-floor SLA of <1 sec/layout and stays under my RAM threshold.

Next steps for industrial adoption:

- Port placement kernel to CUDA (for real GPU speedup on T4 GPUs)
- Experiment with PyPy and Cython to reduce runtime object overhead
- Integrate skyline-based packing and support for shape rotation constraints per-pattern
- Build a REST microservice from the Python engine to enable modular deployment alongside PowerBASIC

This project also gave me the opportunity to apply academic techniques like SA and GA to real manufacturing pipelines showing the immediate ROI of machine learning systems thinking in traditional engineering environments.

XI. Acknowledgments

I sincerely thank the MSML605 Professor Dr. Jerry Wu for his guidance throughout this project. I would like to express my heartfelt gratitude for their invaluable mentorship, critical feedback, and for providing the academic freedom to pursue an industry-grounded problem. His encouragement and structured guidance helped shape this project from a prototype into a validated engineering study and supporting my drive to modernize a mission-critical system.

I also thank my colleagues at my parent company in India, where this challenge originally emerged. Their practical feedback and support during this transition from PowerBASIC to Python-based tooling made it possible to validate our ideas in a real manufacturing setting.

XII. About the Author

Sudharsan Thiyagarajan is currently pursuing his graduate studies in the College of Computer, Mathematical, and Natural Sciences (CMNS) at the University of Maryland, College Park. With a background in manufacturing automation and software development, his professional experience spans working on high-performance layout optimization engines used in the garment industry. He has led the modernization of legacy industrial systems by applying heuristic and metaheuristic algorithms using modern Python-based tooling. His interests include computational geometry, large-scale optimization, industrial systems integration, and accelerating real-time inference with GPU-based solutions. He continues to bridge academic techniques with real-world engineering applications to create scalable, efficient systems for smart manufacturing.

XIII. References

[1] E. G. Coffman, M. R. Garey, and D. S. Johnson, “An application of bin-packing to multiprocessor scheduling,” *SIAM J. Comput.*, vol. 9, no. 1, pp. 36–46,1980.

[2] E. Hopper and B. Turton, “A review of algorithms for 2D rectangular packing problems,” *Eur. J. Oper. Res.*, vol. 141, no. 2, pp. 361–381, 2002.

[3] P. C. Gilmore and R. E. Gomory, “A linear programming approach to the cutting-stock problem,” *Oper. Res.*, vol. 13, pp. 94–120, 1965.

[4] S. Hong, J. Kang, and J. Park, “Simulated annealing-based local search for variable-sized bin packing,” *Comput. Oper. Res.*, vol. 41, no. 5, pp. 120–131, 2014.

[5] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*, Wiley, 1990.

[6] J. Christiaens and G. Van den Berghe, “Ruin and recreate strategies for vehicle routing,” in *Proc. GECCO*, 2020.

[7] E. K. Burke and Y. Bykov, “Late acceptance hill-climbing,” *J. Oper. Res. Soc.*, vol. 68, pp. 1–11, 2017.

[8] D. K. Friesen and M. A. Langston, “Variable sized bin packing,” *SIAM J. Comput.*, vol. 15, no. 1, pp. 222–230, 1986.

[9] Y. Wei, B. Liu, and Y. Cui, “Goal-driven binary search heuristic for two-dimensional variable-sized bin packing,” *Comput. Oper. Res.*, vol. 40, no. 10, pp. 2448–2456, 2013.

XIV. GitHub Detail

sdshan99/NestingAnalysisPBvsPython