Introduction

Overview

This paper discusses the general hardware placement problem, and the 2 categories of methods used to optimize for it. Those are the iterative methods, and the analytical methods.

This paper starts by breaking down the goals of the work described here, followed by a description of the placement problem. An in depth discussion about the 2 categories of placement methods, and the ones implemented follows. This paper ends with the experiment results, as well as a high level analysis on the effectiveness of these methods.

The experiment output can be found in the appendix section. The source code is available on Github https://github.com/lennoxho/CSC466-Project.

Goals

This paper aims to guide the computer scientist through one of the hardest challenges in the electrical engineering field - the placement problem. This is a suitable problem to be studied by computer scientists, since it can be easily reduced to a directed, weighted network (graph) problem. As such, numerical optimization methods, which are within the realm of study in computer science, can be employed.

The work described here will hopefully motivate the computer science reader to perform further research on the problem described.

Background

Basics

Before the placement problem can be described, a few key concepts and terms must first be introduced. Entry level knowledge of digital logic is expected. Terms such as **logic gates**, **flip-flops**, **input and output pins**, as well as **wires** should be familiar to the reader.

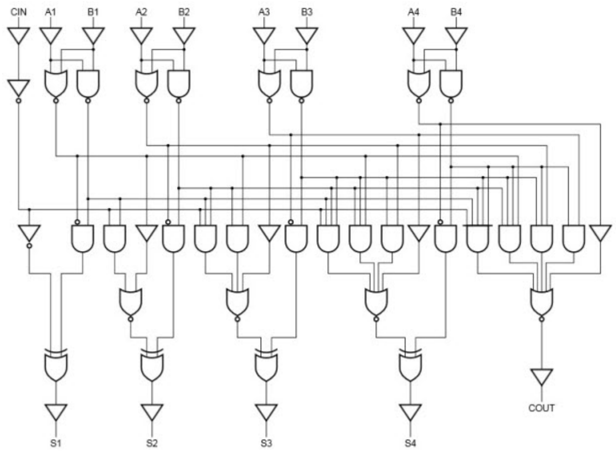


Figure 1: Carry Adder[1]

The diagram above describes a simple 4-bit carry lookahead adder. Given 2 4-bit input signals, the circuit will produce the sum of those inputs. This output can then be connected to a FIFO to be used in another operation.

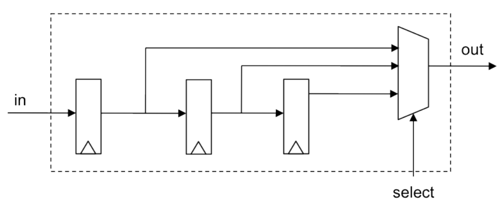


Figure 2: FIFO [2]

Note that in both diagrams presented so far, the **clock** signals are not displayed. This is because in contemporary hardware design, clock signals are driven globally by chip intrinsics, and do not share the same hardware resources (gates, flip-flops, wires etc.) as the **"core"** logic ("core" is the term usually used to describe user-specified parts of the circuit). As such, the work here will only consider the placement problem in terms of core logic.

Also note that in the first diagram, some of the gate outputs branch out. This is referred to as **fanout**. Each output port (not to be confused with output pin) from a gate, flip-flop etc. can have zero, one or more fanouts. On the other hand, each input port can only have 1 **fanin**. Since each hardware component can have multiple inputs and outputs, the total number of fanins and fanouts can be huge.

Usually, **electronic design automation (EDA)** tools do not work with individual gates or similarly small hardware elements. Instead, they tend to work with larger building blocks such as **look up tables (LUTs)** and **registers**. A LUT, as its name implies, can emulate any gate or logic based on its configuration, while a register is a bank of flip-flops strung together. This LUT-register configuration (also know as **adaptive logic module** or **ALM**) is most commonly used in **field programmable gate arrays (FPGAs)** as their fundamental building blocks.

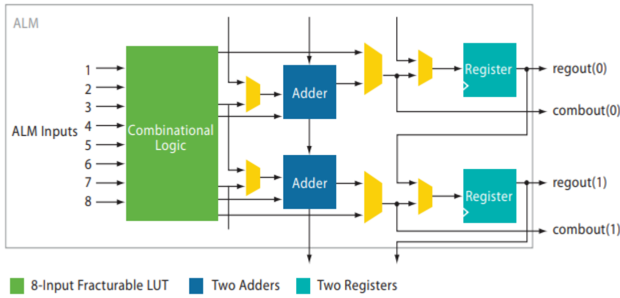


Figure 3: Altera$^{\text{TM}}$ ALM[3]

Note that in the diagram above, adders are included as an (optional) optimization feature.

Chip

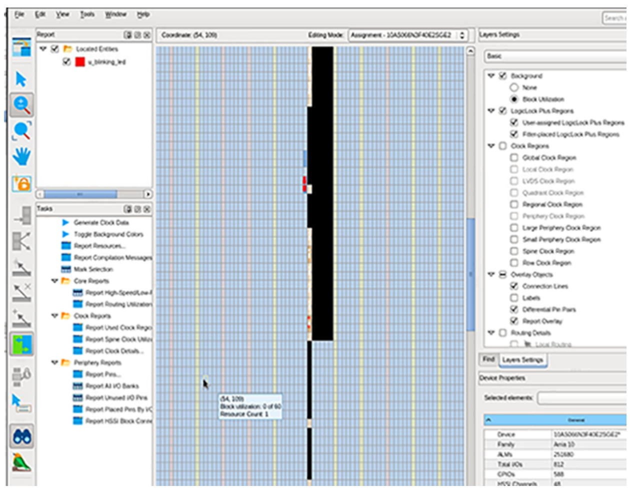


Figure 4: Quartus$^{\text{TM}}$ Chip Planner[4]

A **chip** is the physical die that hardware components sit on. Chips are usually organized in a grid manner, where each (rectangular) **cell** is a location where a component can be placed. The chip also provides power and connects each pair of cells with wires.

At the edges of the chip, there are input and output (IO) pins. Given some input, the signals are routed through some logic on the chip, before exiting using the outputs.

Usually, each cell on a chip can only be used to for a subset of all available components, due to power, congestion or clock network proximity.

Note that contemporary chips usually have a 3^{rd}/z dimension, by layering components on top of one another. This configuration was not considered in this paper, although it is interesting to note that the 3-dimensional problem can be reduced to a 2-dimensional one.

Model used in Experiments

For the purposes of this paper, the placement problem will be considered in terms of only 2 hardware elements - LUT and register (referred to in the source code as flip-flop/"FF"). While this is a huge simplification over the actual placement problem, it is still a reasonable one. In fact, modern EDA tools usually make use of such high level abstractions to speed up the optimization process. For the rest of this paper, each of those hardware elements will be referred to as an **"atom"**

The most obvious data structure to model a circuit would be a directed graph. In fact, since every edge can have more than 2 endpoints, this is a more specific type of graph - a **netlist**. The term **"net"** is used to describe an edge with more than 2 endpoints. Hence the term "netlist".

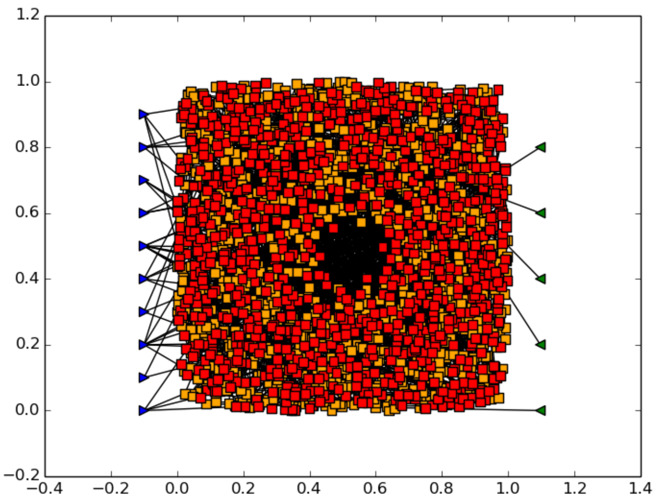


Figure 5: Random netlist with 1000 LUTs and 1000 Registers, 10 input pins, 5 output pins in 1 phase

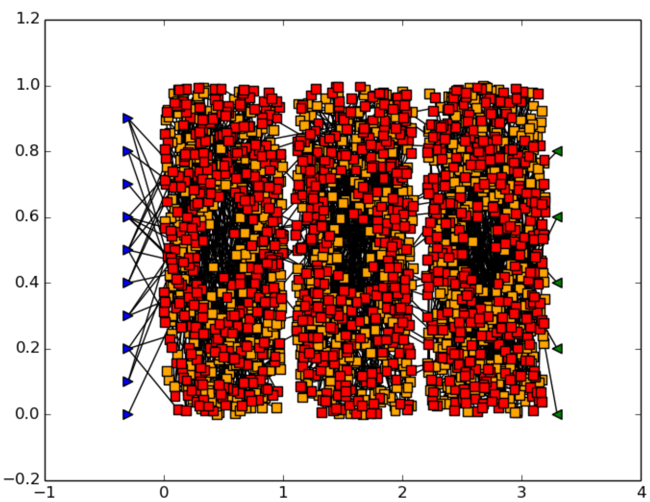


Figure 6: Random netlist with 1000 LUTs and 1000 Registers, 10 input pins, 5 output pins in 3 phases

Note that the second diagram had the atoms organized in 3 "phases". In real designs, atoms are usually organized in reusable "modules", which may prove suboptimal for certain types of placement methods.

Those netlists were generated by the `run_placer` executable and plotted using the `src/draw_netlist.py` Python script.

For the chip, a grid configuration was used, where each alternating cell can hold either a LUT or a register.

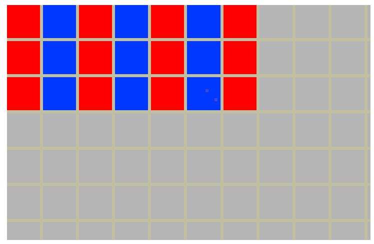


Figure 7: Model Chip

The red squares represent cells that can hold LUTs, while the blue cells can hold registers. The grey cells were not colored for brevity.

The Placement Problem

Given a netlist, which represents the logical configuration of a circuit, as well as a chip, the task is to map the former to the latter, while minimizing the **critical path** between the input and output pins (which are on the edges of the chip). The critical path determines the maximum frequency the circuit can operate under, which directly corresponds to chip performance. This is called the **placement problem**, which is very similar (can be reduced) to the traveling salesman problem.

Since the traveling salesman problem is a well know NP-hard problem, a brute force approach to optimize the placement problem would be ill-advised, especially when a real circuit usually has hundreds of thousands or even millions of atoms to place. This is why placement methods, which approximately solve the placement problem, were devised. As mentioned earlier, there are 2 main categories of placement methods.

Critical Path

The **critical path** is the length of the longest path between any input and output pins. The path here refers to the physical routed wire. Since the **routing problem**, which is out of scope of this paper, is usually solved after the placement problem, other distance metrics are used instead.

Iterative Placement Methods

In iterative placement methods, the distance metric is usually measured by using the **half-perimeter wire-length (HPWL)** distance, also known as the **bounding box (BBOX)** distance.

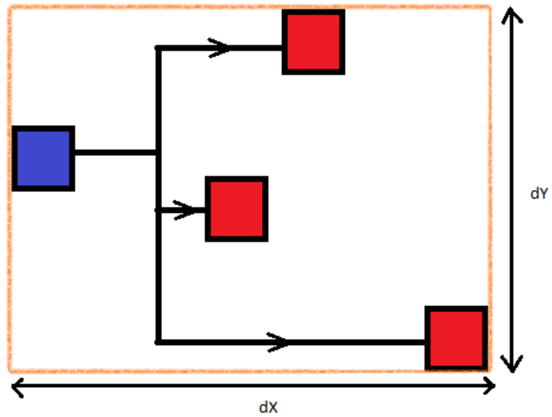


Figure 8: BBOX

The BBOX distance is measured on a per-net basis, by finding the smallest box that would fit all nodes connected to the net. In the example above, the BBOX distance is measured to be $BBOX = dX + dY$. The sum of critical paths for the entire design is then approximated to be $\sum_{n \in all\ nets} BBOX(n)$.

Since physical wires on a real chip usually travel in a grid-like manner, the $BBOX$ distance is a good metric for the critical path distance.

Randomized placement

```
for atom in netlist:
    place atom in unoccupied location (x,y) on chip

L = 0
for net in netlist:
    L += BBOX(net)

L_prev = inf
for i in [0, ITER_MAX]:
    L_prev = L

    type = random_type(netlist)
    atom_a = random_atom(chip, type)
    atom_b = random_location(netlist, type)
    swap(atom_a, atom_b, chip)

    L_new = update_BBOX(atom_a, atom_b)
    if L_new < L_prev:
        L = L_new
    else:
        unswap(atom_a, atom_b, chip)
```

As its name implies, the randomized iterative algorithm randomly chooses 2 atoms (or empty cells) of the same type at each iteration, swap their locations, and compute the new $BBOX$ distance. If an improvement is achieved, the swap is kept. Otherwise, the atoms (or empty cells) are swapped back to their previous locations.

Note that although the atoms are randomly chosen, the critical path distance will never increase, that is, it monotonically decreases. This is due to the swap-backs that are performed when the $BBOX$ distance increases.

Simulated annealing

An issue with the randomized iterative algorithm, is its inability to "climb" out of local minima, due to its monotonically decreasing nature. The simulated annealing method is a slight modification to the randomized algorithm to add the ability to take a bad swap for a chance of getting to a lower local minimum.

```
for atom in netlist:
    place atom in unoccupied location (x,y) on chip

L = 0
for net in netlist:
    L += BBOX(net)

T = HOT
L_prev = inf
for i in [0, ITER_MAX/num_swaps_per_temp]:
    for j in [0, num_swaps_per_temp]:
        L_prev = L

        type = random_type(netlist)
        atom_a = random_atom(chip, type)
        atom_b = random_location(netlist, type)
        swap(atom_a, atom_b, chip)

        L_new = update_BBOX(atom_a, atom_b)
        if L_new < L_prev:
            L = L_new
        else:
            if uniform_random(0, 1) < exp((L_new - L_prev)/T):
                L = L_new
            else:
                unswap(atom_a, atom_b, chip)

    T *= COOLING_FACTOR
```

The main difference between simulated annealing and the randomized method is the introduction of a temperature, T. The main idea is that while the temperature is still "HOT", then `exp((L_new - L_prev)/T)` will produce a larger number in $(0, 1]$. This translates to a higher probability of "going up the hill" during early iterations. During later iterations, as the temperature "cools" down, less "risk" is taken. This method works well in practice, provided good calibration.

Analytical Placement

A slight variation of the Euclidean distance is usually used for analytical placement methods. The difference is due to the multi-fanout nature of the netlist.

Given a net, the Euclidean distance is calculated by first replacing the net with straight lines between each pair of atoms.

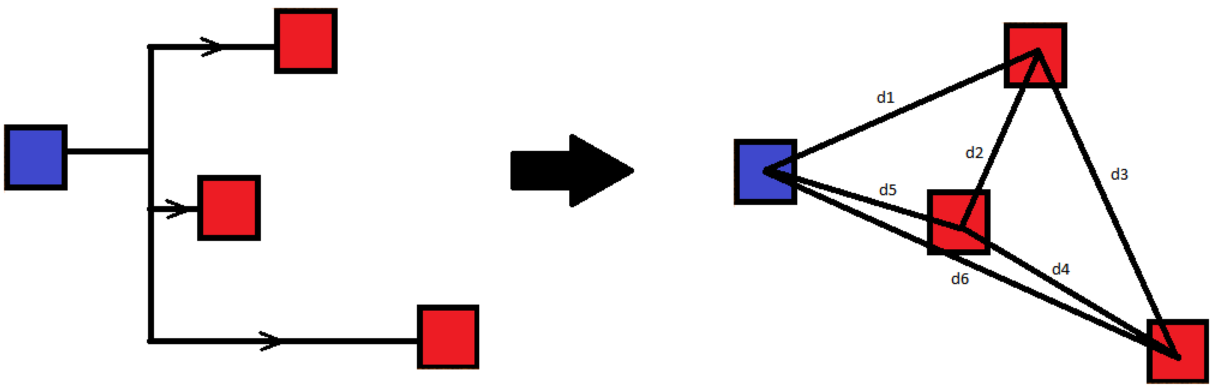


Figure 9: Euclidean distance for net

The total distance is then calculated by

$$L = \frac{\sum_{i=1}^{N} d_i}{N-1}$$

Where $\quad d_i = (a_x - b_x)^2 + (a_y - b_y)^2 \quad , a, b$ are endpoints of d_i

Quadratic Placement

The observation that leads to the quadratic placement method is the fact that the naive algorithm is NP-hard exactly because it is attempting to solve the discrete placement problem. By relaxing the discrete problem to a continuous one, other polynomial-time methods may be considered. A key idea is the opportunity to optimize the x and y components of the equation separately when the Euclidean distance is used as the distance metric.

Given an atom k which is connected to the set of atoms A and the set of pins P,

$$L_x = \Big(\sum_{a \in A} w_a (x_k - x_a)^2\Big) + \Big(\sum_{p \in P} w_p (x_k - x_p)^2\Big)$$

w_i is the weight of the edge between k and i

$$L'_x = 2\Big(\sum_{a \in A} w_a (x_a - x_k)\Big) + 2\Big(\sum_{p \in P} w_p (x_p - x_k)\Big)$$

Note that the weights, $w_i = \frac{n}{N-1}$, where N is the number of fanout for the corresponding net, and n is the number of connections between atoms k and i.

Since the equation above is a 2^{nd} degree polynomial, its first derivative exists and is continuous.
Also, since the equation is positive, its critical point is a minimum.

$$\begin{aligned}\text{Let } L'_x &= 0\\ 0 &= \Big(\sum_{a\in A} w_a(x_a - x_k)\Big) + \Big(\sum_{p\in P} w_p(x_p - x_k)\Big)\\ \Big(\sum_{b\in A\cup P} w_b\Big)x_k - \Big(\sum_{a\in A} w_a x_a\Big) &= \sum_{p\in P} w_p x_p\end{aligned}$$

Note that the components involving pins are separated to the right, since the location of pins are fixed. That is, those components are constants.
Repeating the derivation above for all atoms, the following linear system can be constructed.

$$\begin{pmatrix} A_1 & -w_{12} & -w_{13} & ... & -w_{1n}\\ -w_{21} & A_2 & -w_{23} & ... & -w_{2n}\\ ... & ... & ... & ... & ...\\ -w_{n1} & -w_{n2} & -w_{n3} & ... & A_n \end{pmatrix} \begin{pmatrix} x_1\\ x_2\\ ...\\ x_n \end{pmatrix} = \begin{pmatrix} B_1\\ B_2\\ ...\\ B_n \end{pmatrix}$$

$$\begin{aligned} A_i &= \sum_{b\in A\cup P} w_{ib}\\ B_i &= \sum_{p\in P} w_{ip}x_p\\ w_{ij} &= \text{weight of the edge between atoms i and j.}\\ &\quad\ \text{0 if does not exist}\end{aligned}$$

The same process can be repeated for the y components. By solving those those linear systems, the placement that minimizes the Euclidean distance can be obtained, which should translate to good placement performance. Note that in the 3-dimensional problem, a linear system for the z components can similarly be created.

The constructed linear systems are **symmetric**, since $w_{ij} = w_{ji}$. The linear system can also be equivalently expressed as a minimization problem:

$$Ax = b \iff \min_{x\in R^n} \frac{1}{2}x^T Ax + b^T x$$

Assuming A is not symmetric positive semi-definite,

$$\exists z \in R^n, \quad z^T Az < 0$$
$$\text{Then} \quad f(tz) = \frac{1}{2}t^2 z^T Az + tb^T z \to -\infty \text{ as } t \to \infty$$

The last deduction contradicts the earlier assumption that $f(x)$ has a minimum. Therefore, the constructed linear system is also **positive semi-definite**. With **symmetric positive semi-definite** linear systems, techniques such as the **conjugate gradient method** can be used to efficiently solve them.

However, the resulting solutions are unlikely to contain whole numbers, which is required by the grid nature of the chip. There is also the issue of congestion, as the quadratic method tends to place atoms close together. Recursive partitioning is used to overcome those limitations.

Recursive Partitioning

The simplest partitioning scheme is to **partition** the chip into 2 halves, and assign half of the atoms to each. The choice of atoms for each partition can be chosen by sorting the atoms by x then y coordinates (from quadratic placement), and assigning the first half to the first partition, and the rest to the second partition.

2 straightforward approaches to splitting the chip are by bisection, and by an adaptive method. Bisection simply splits the chip along the dimensions of the chip. The adaptive approach splits the chip according to the location of the median atom.

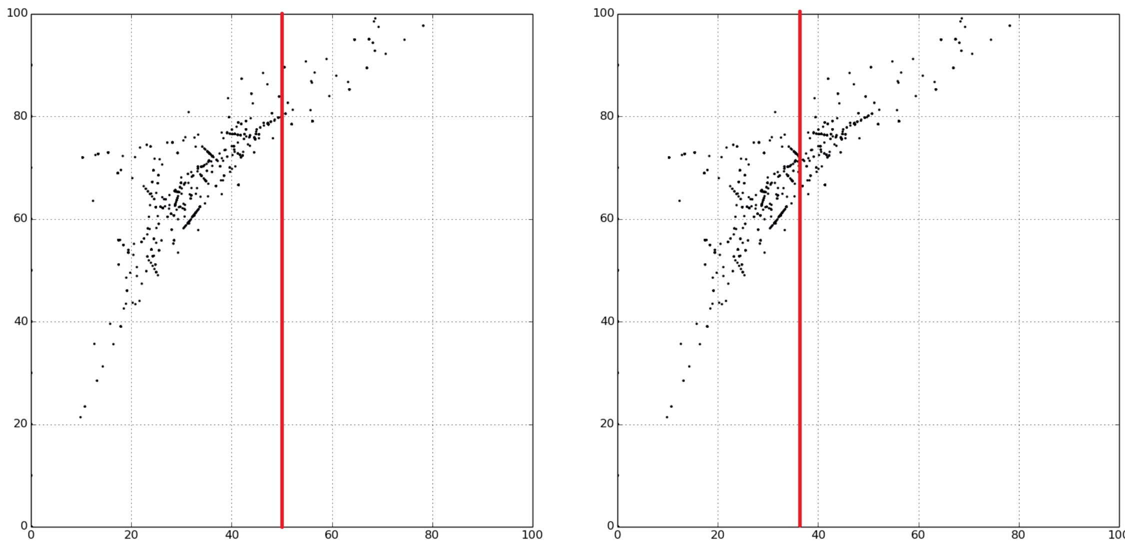


Figure 10: Bisection (left) and Adaptive (right) partitioning

The quadratic placement and partitioning steps can be repeated on smaller and smaller subpartitions until a good spread is obtained. This process is called **relaxation**.

After relaxation is performed, the atoms still need to be assigned to actual grid locations within their respective partitions. This final step is called **legalization**.

Choice of Algorithms

Instead of attempting to directly solve the placement problem, the random placement and simulated annealing methods make incremental improvements to the result. There are a few key advantages to this approach. The first is the computation cost involved. It was noted earlier that solving the placement problem directly would had been a NP-hard problem. Iterative methods on the other hand only incur polynomial costs at each iteration. This is at the expense of accuracy and convergence, since both are dependent on the number of iterations and probabilistic models. Since complete accuracy is never a goal of hardware designers, those trade-offs are usually acceptable. The other advantage of iterative methods, is the incremental aspect of them. A hardware designer may choose to limit the number of iterations, or stop iterating when the critical path distance reaches a certain threshold. Each incremental snapshot is a legal placement, where when solving the problem directly, one would not have that option.

As noted earlier, the quadratic placement method sacrifices accuracy for polynomial computation time. This relaxation approach is very similar to how integer programming problems are usually transformed into linear programming problems in exchange for accuracy. The reader may have noted that the quadratic placement method can be reformulated as a quadratic programming algorithm, hence the similarity. The strengths and weaknesses of the quadratic programming algorithm therefore applies, most notably is its general lack of accuracy (after legalization). The lack of accuracy is not a big concern to hardware designers, since the role of quadratic placement methods are usually to increase the spread (and therefore reduce congestion), while not sacrificing performance.

Implementation

All methods and techniques described above had been implemented in C++ with heavy use of the Boost libraries[5]. The Eigen library[6] was also used to perform computations involving linear algebra. Due to the lack of similar EDA open source projects, all code posted on the Github page had been written from scratch. The executable `run_placer` will reproduce the experiments described later in this paper.

The "Robust Cholesky decomposition with pivoting" `ldlt`[7] linear solver in Eigen was used in the quadratic placement method for its superior performance with symmetric positive semi-definite matrices.

Three additional Python scripts were also created: `src/draw_netlist.py`, `src/plot_iterations.py` and `src/plot_snapshots.py` to visualize the created netlists, plot the iterations of iterative methods, and plot the progression of placement methods respectively.

The instructions to build and run those executables and scripts can be found on the Github page.

Experiment Results

The BBOX distance was used as the metric in determining the performance of the final placement. The lower the final BBOX distance, the better the placement.

Visualizations

The diagrams below were produced by running the various placement methods on the 1 phase netlist in figure 5.

For iterative placement methods, all atoms were initially placed sequentially on the left.

Random Placement

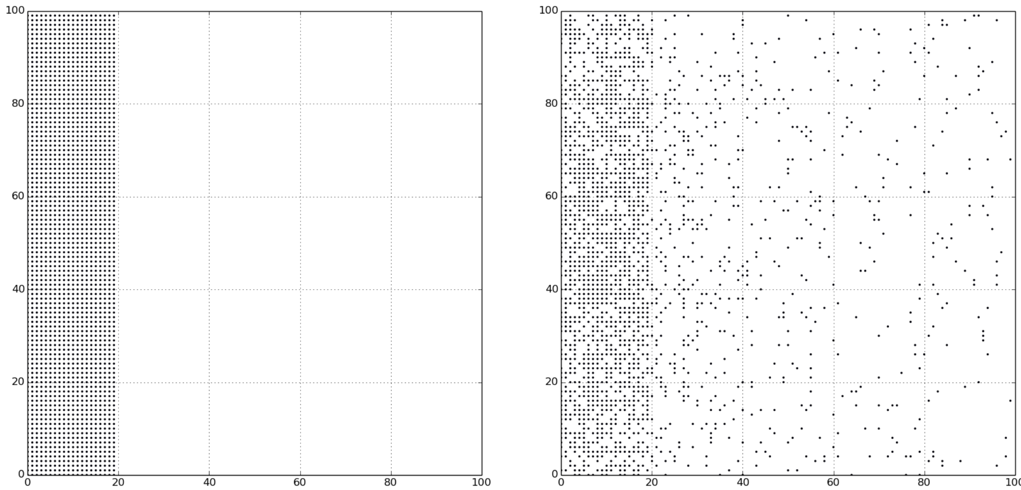


Figure 11: Random placement snapshots: before (left), after (right)

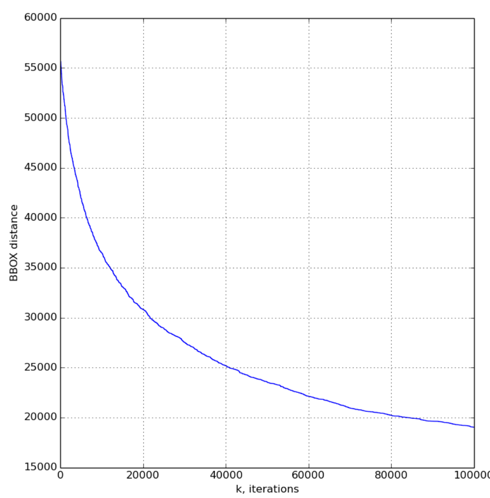


Figure 12: Random placement iterations

Simulated Annealing

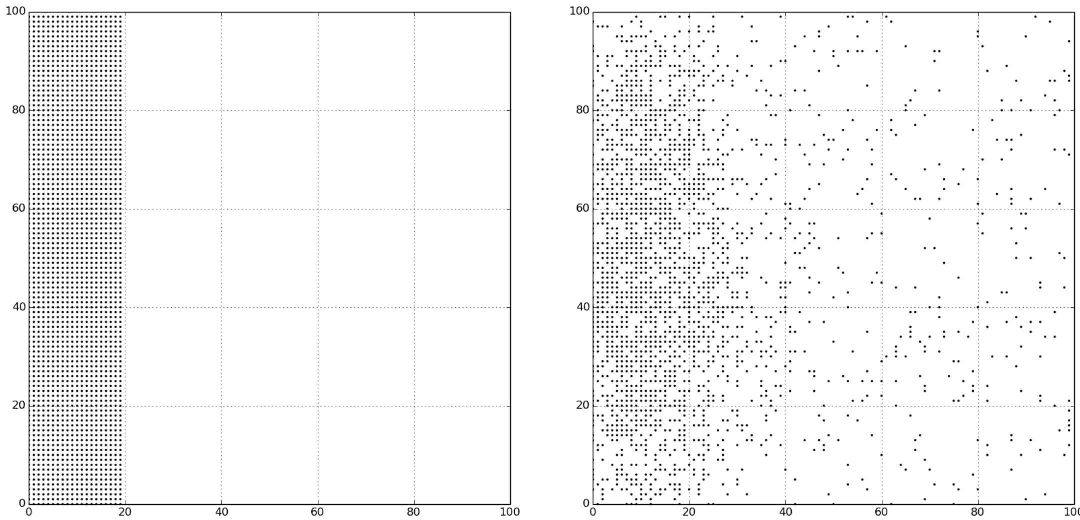


Figure 13: Simulated annealing snapshots: before (left), after (right)

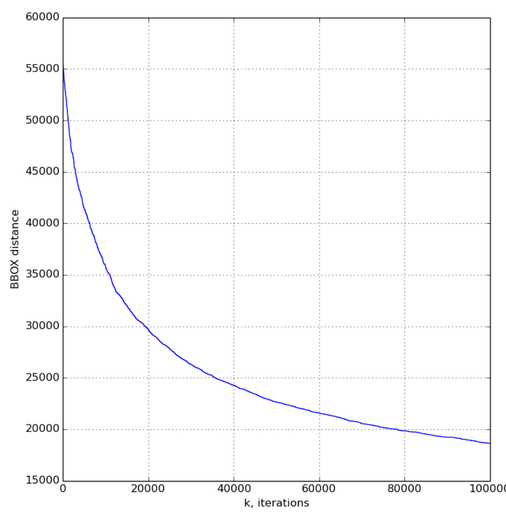


Figure 14: Simulated annealing iterations

For the Quadratic Placement method, the chip was split across the x-axis during the 2^{nd} pass, and the y-axis during the 3^{rd} pass.

Quadratic Placement with Bisection Partitioning

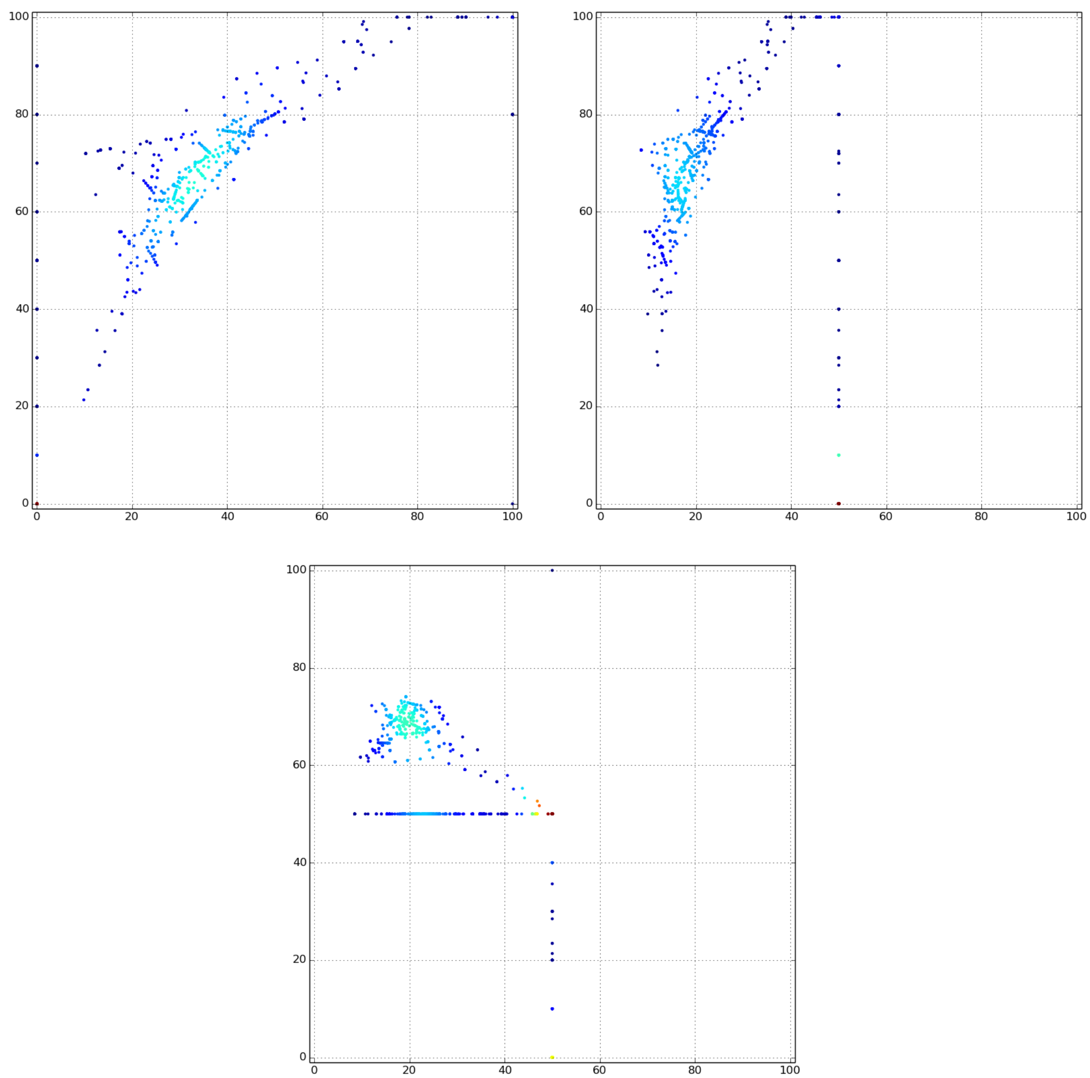


Figure 15: Quadratic Placement with Bisection Partitioning snapshots $1 \rightarrow 2 \rightarrow 3$ pass

Random Placement after Quadratic Placement with Bisection Partitioning

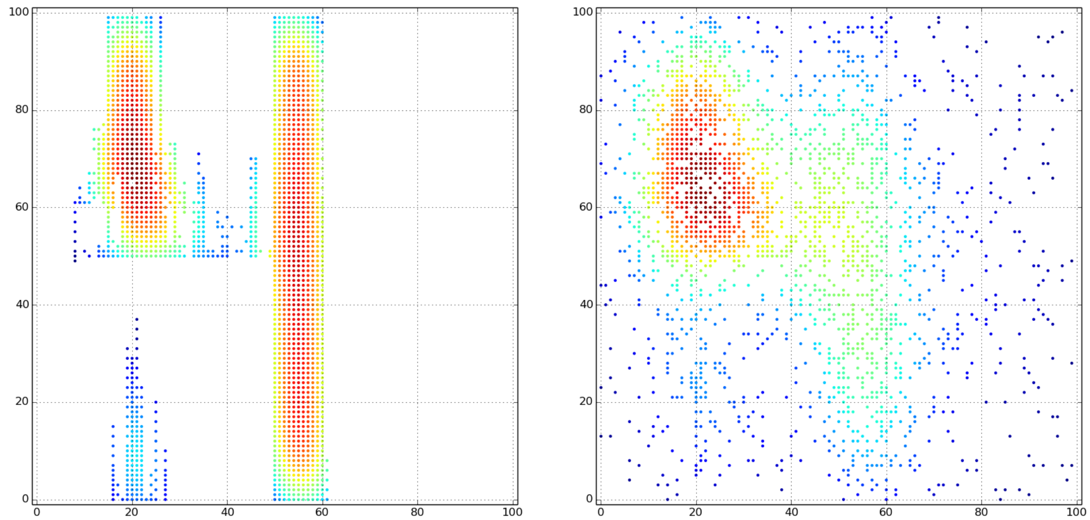


Figure 16: Random Placement after Quadratic Placement with Bisection Partitioning snapshots: before (left), after (right)

Simulated Annealing after Quadratic Placement with Bisection Partitioning

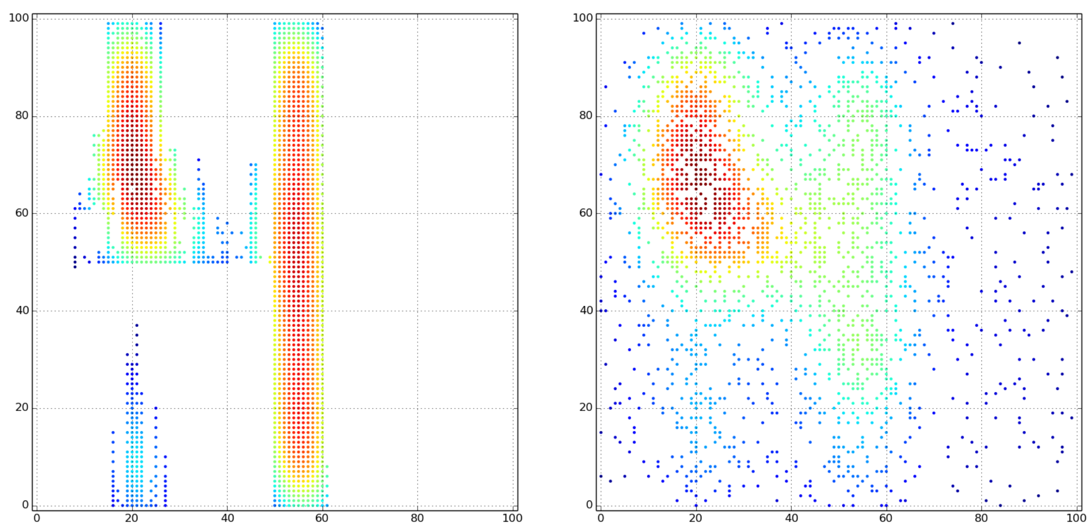


Figure 17: Simulated Annealing after Quadratic Placement with Bisection Partitioning snapshots: before (left), after (right)

Quadratic Placement with Adaptive Partitioning

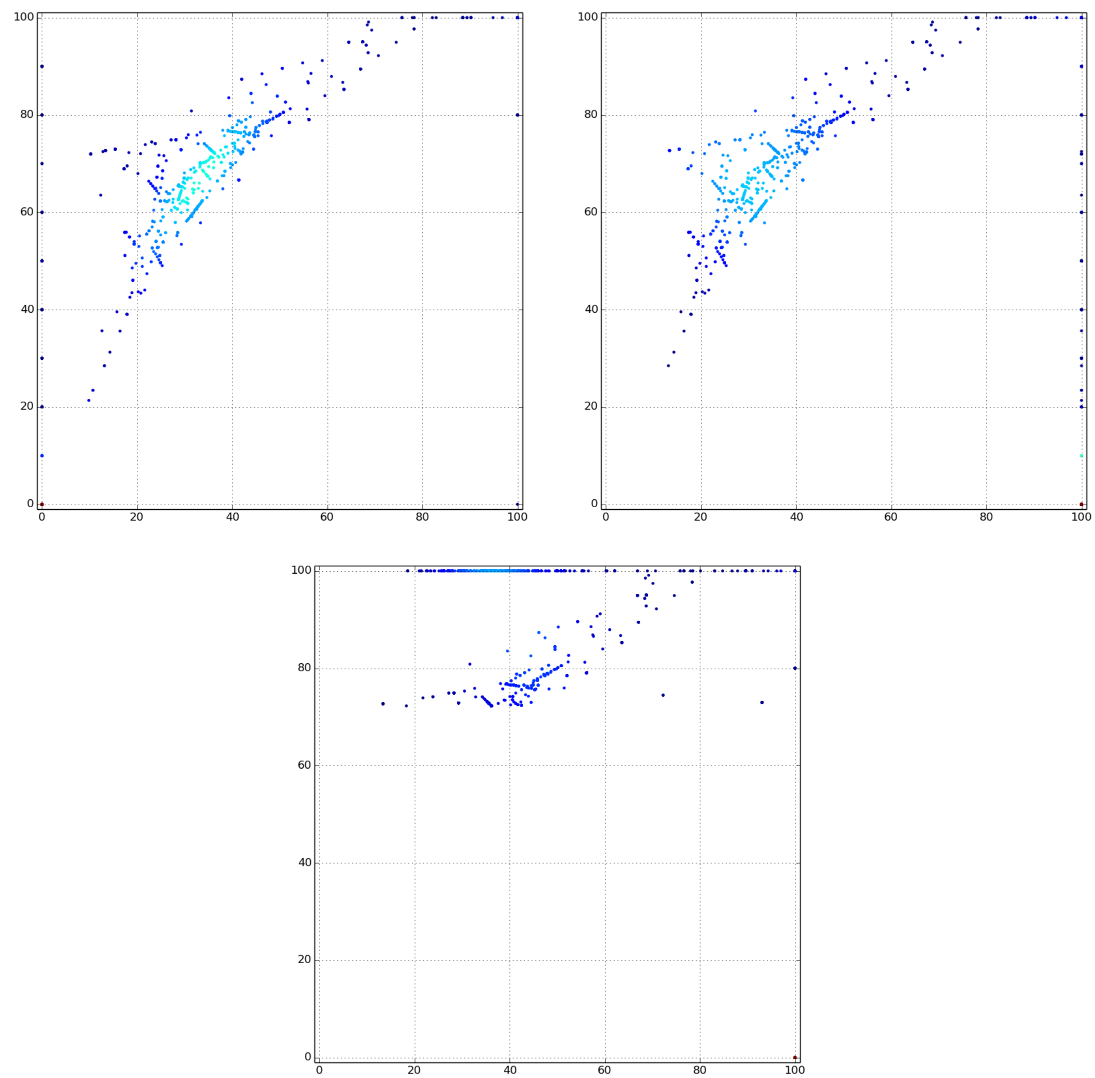


Figure 18: Quadratic Placement with Adaptive Partitioning snapshots 1 → 2 → 3 pass

Random Placement after Quadratic Placement with Adaptive Partitioning

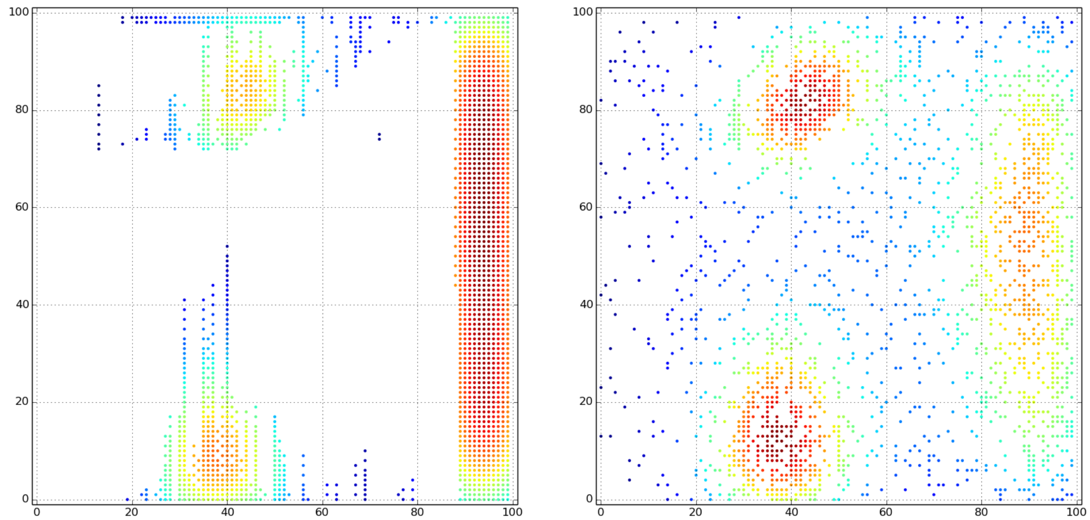


Figure 19: Random Placement after Quadratic Placement with Adaptive Partitioning snapshots: before (left), after (right)

Simulated Annealing after Quadratic Placement with Adaptive Partitioning

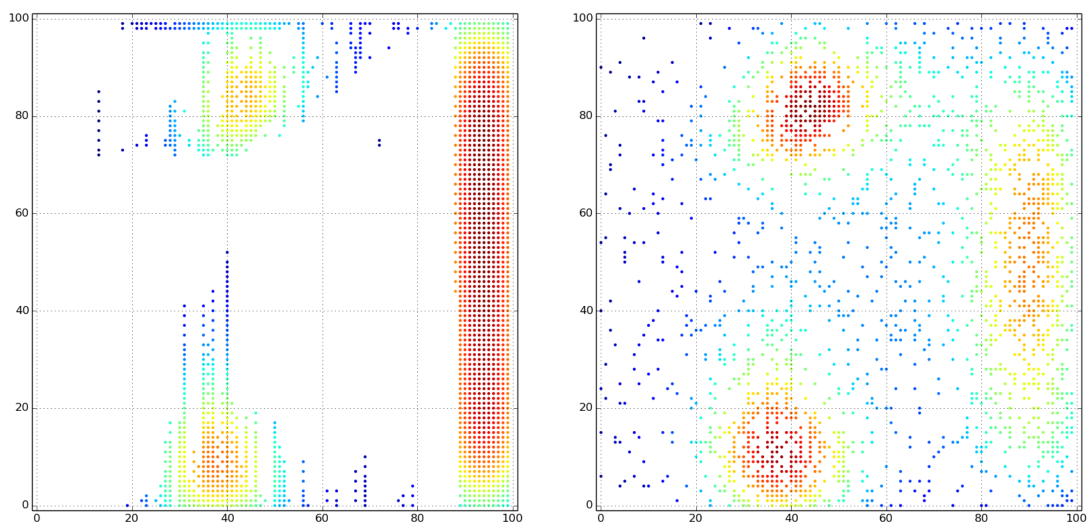


Figure 20: Simulated Annealing after Quadratic Placement with Adaptive Partitioning snapshots: before (left), after (right)

The diagrams below were produced by running the various placement methods on the 3-phase netlist in figure 6.

Quadratic Placement with Bisection Partitioning for 3 phase netlist

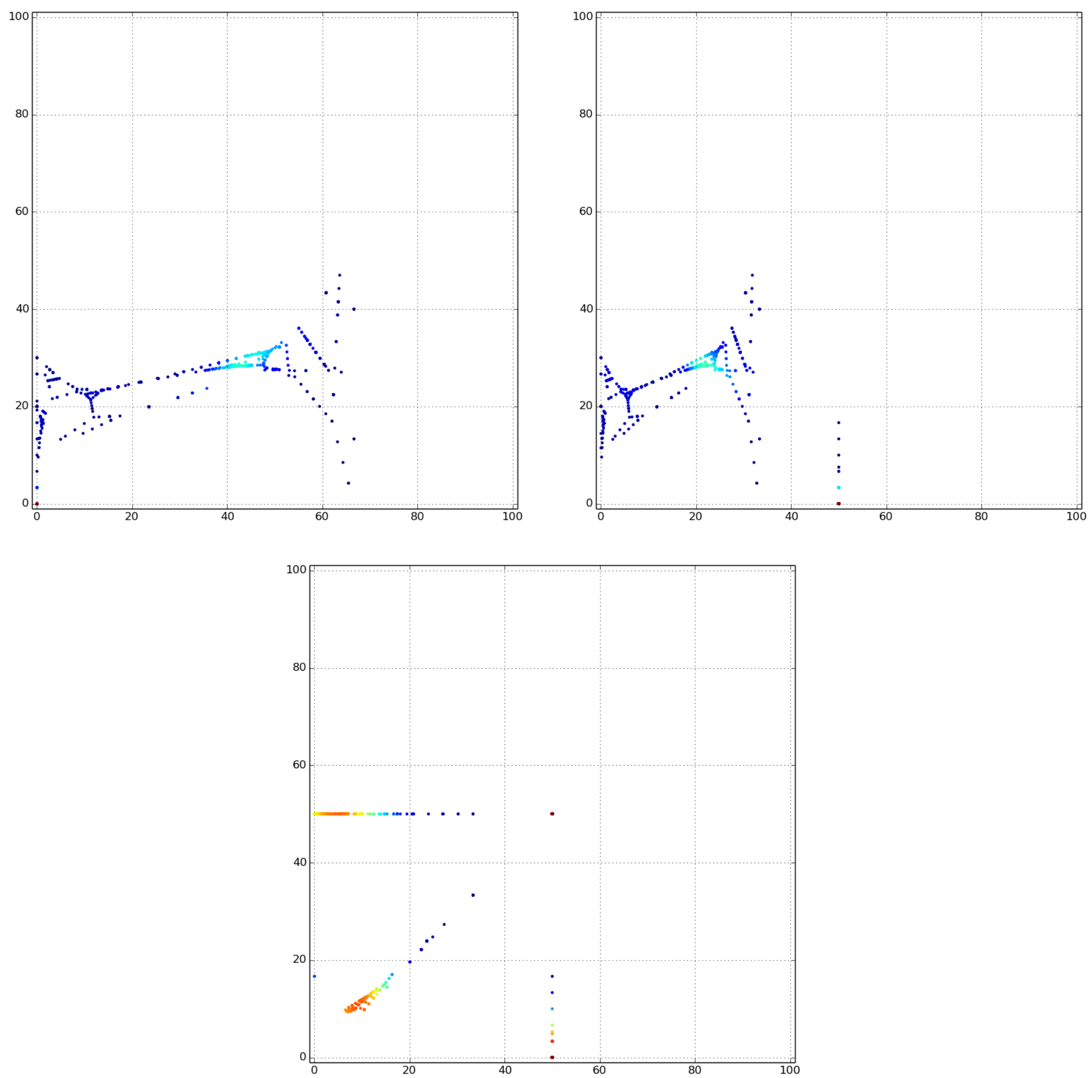


Figure 21: Quadratic Placement with Bisection Partitioning for 3 phase netlist snapshots 1 → 2 → 3 pass

Quadratic Placement with Adaptive Partitioning for 3 phase netlist

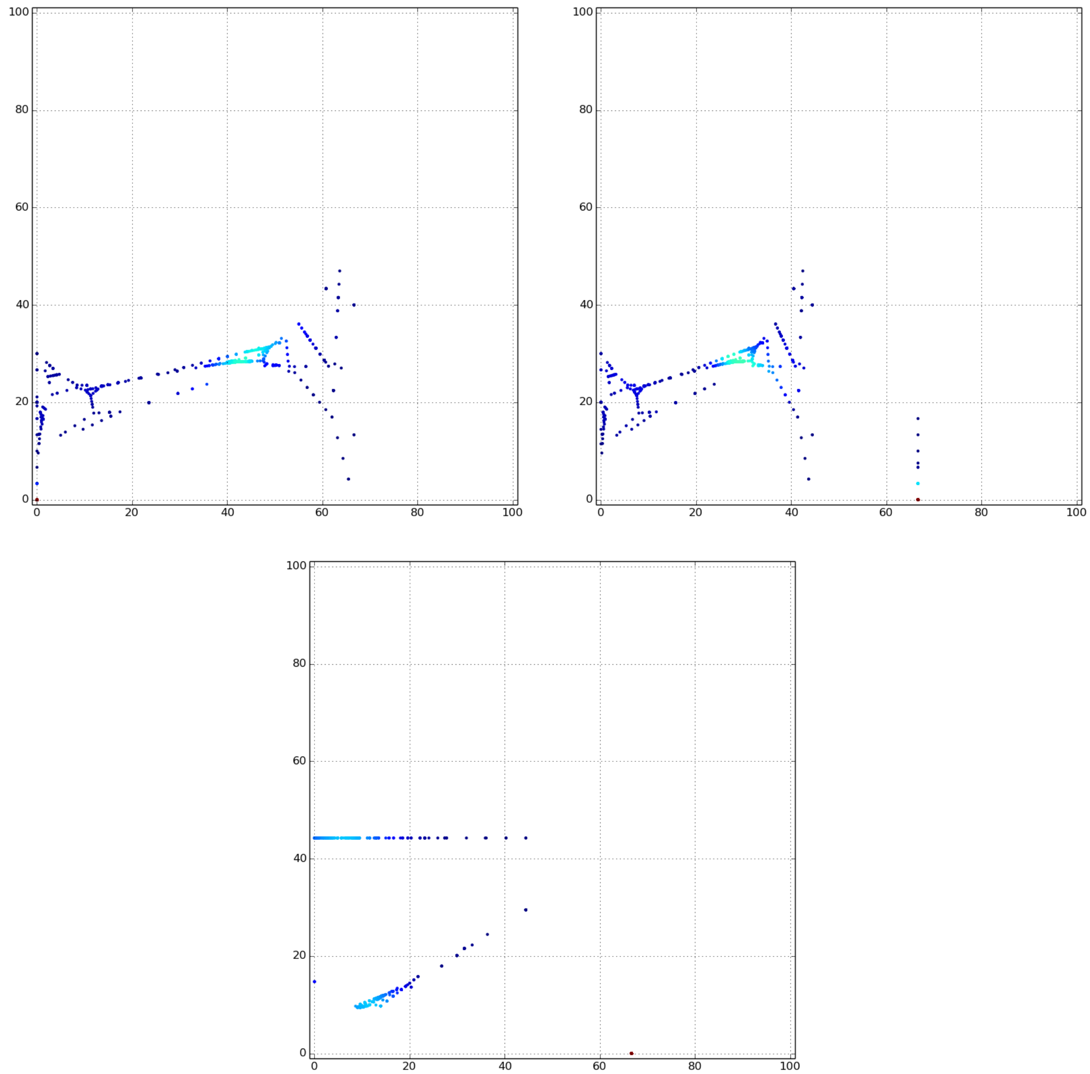
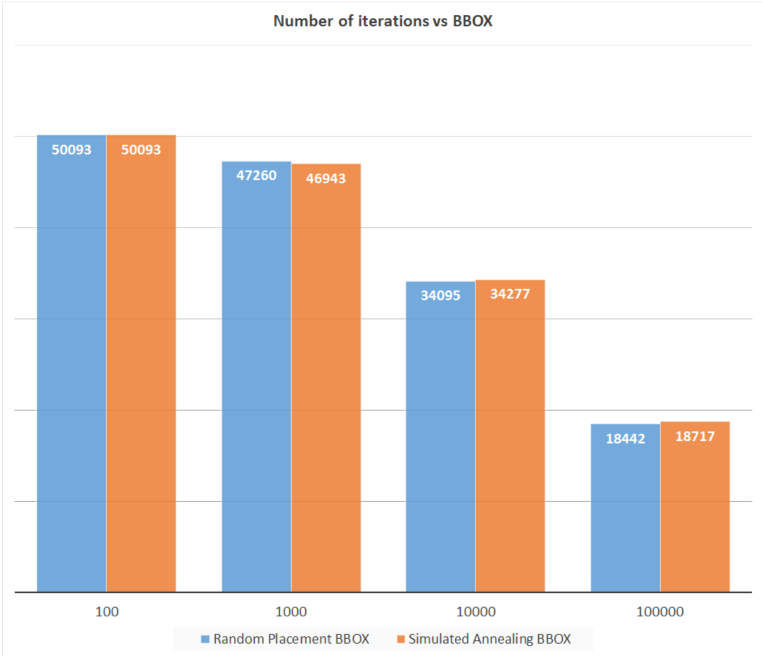


Figure 22: Quadratic Placement with Adaptive Partitioning for 3 phase netlist snapshots 1 → 2 → 3 pass

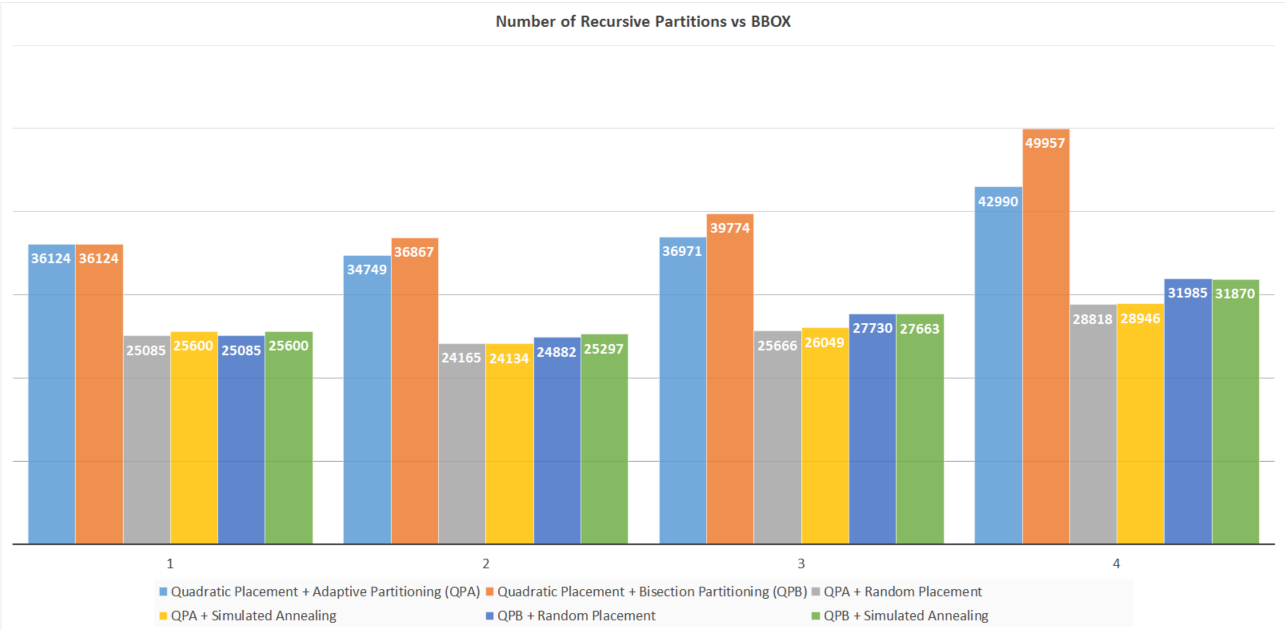
Number of Iterations for Iterative Methods

These data were collected on the 3-phase netlist in figure 6.



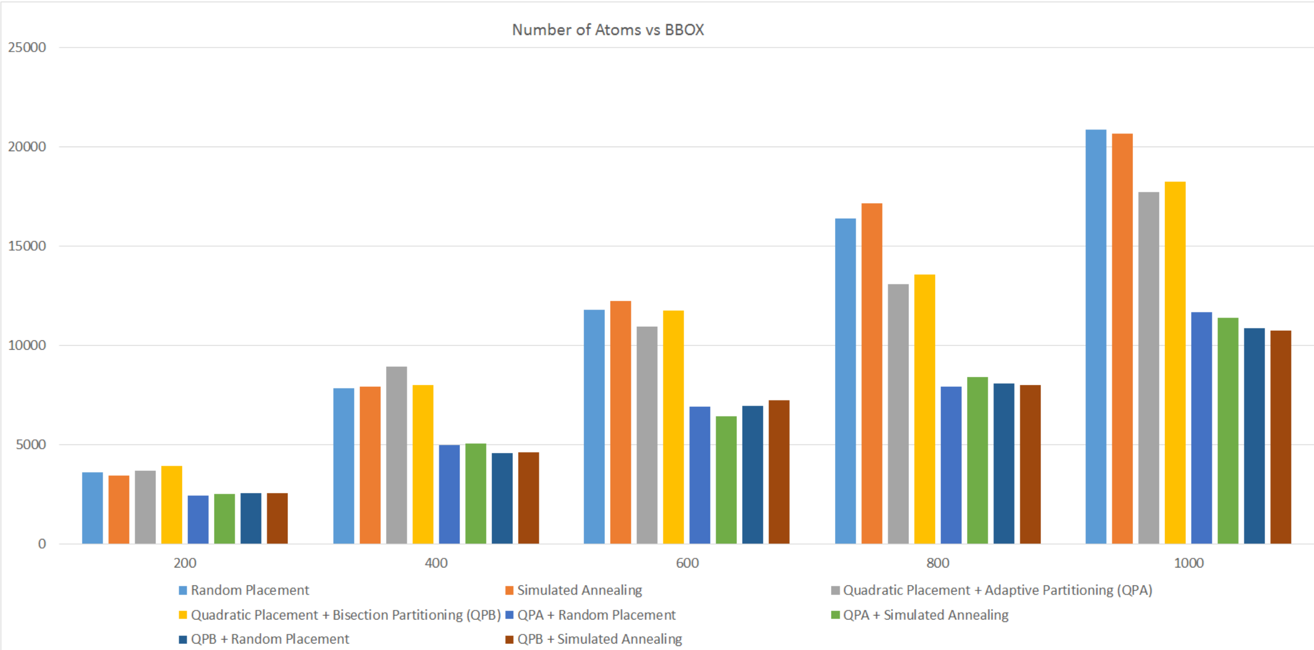
Number of Recursions for Quadratic Placement

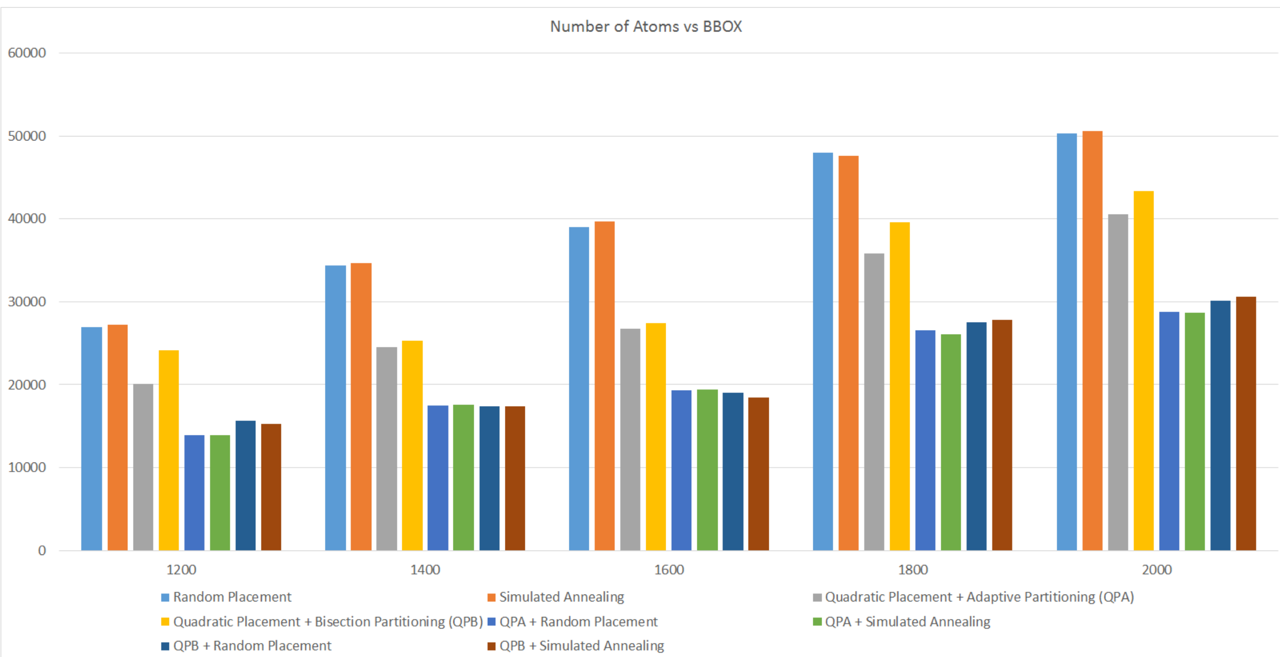
These data were collected on the 3-phase netlist in figure 6.



Number of Atoms

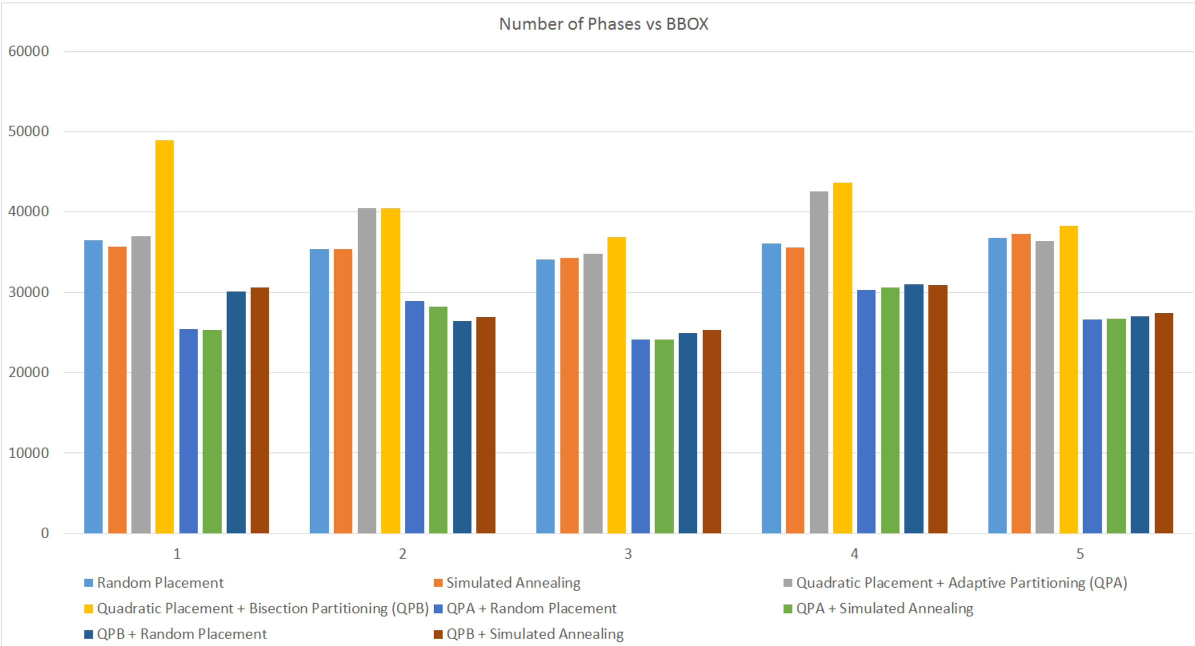
The number of phases was locked at 3 phases for these results.

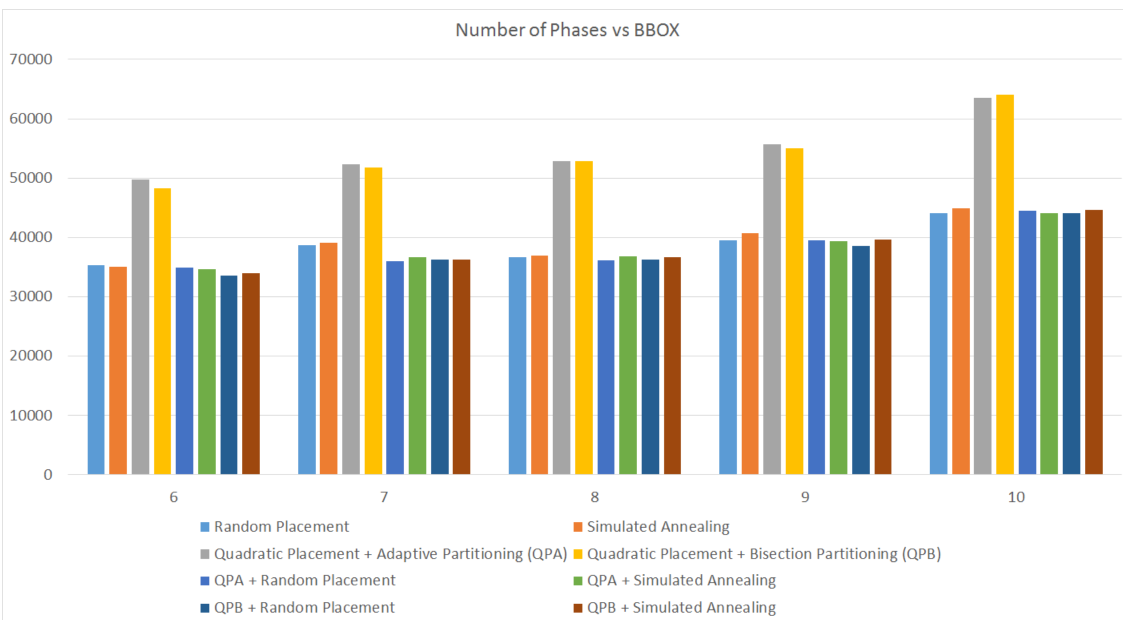




Number of Phases

The number of atom was locked at 2000 for these results.





Analysis

From the experiment results presented, it would seem that as the number of iterations increases, the BBOX distance for iterative methods decreases. This is in line with the understanding that the random placement method will always reject bad swaps, while the simulated annealing method tends to reject them.

The difference in performance of the random placement method compared to the simulated annealing method was negligible. While some of this was due to the lack of good calibration with simulated annealing, the main issue was the low number of constraints imposed in the model. With a higher number of atom types and location constrains, the random placement method would have more difficulty in finding good swaps, while the simulated annealing method would had fared better.

It would also appear from the results that the higher the number of recursive partitions, the worse the performance of the quadratic placement. This was expected, since the BBOX distance is a global measure of performance, performing smaller local optimizations naturally hurt the final BBOX distance. As noted earlier, the reason why quadratic placement methods are still prevalent in EDA tools is due to its good balance between increasing spread (and reducing congestion), and minimizing the critical path. This fact was apparent by observing the performance when running iterative placement methods on the results of quadratic placement.

The number of atoms experiment showed that as the number of atoms increases, iterative methods progressively performed worse. This was due to the increase in resource contention. In other words, the number of placement constraints increases as the number of atoms increases. This increases the probability of bad swaps.

The choice of adaptive versus bisection methods of partitioning seemed to have minor effects on the final BBOX distance. However, it would seem that bisection had a small advantage over the adaptive method, especially for higher atom counts.

Conclusion

From the experiments above, it would appear that a combination of quadratic placement methods followed by iterative methods produces the best results. While the simulated annealing implementation presented did not perform well, it should in theory perform better with good calibration efforts and a more realistic optimization model to work on.

Checklist

1. Create abstractions for a netlist.
2. Write function to produce randomized netlists for benchmarking.
3. Create visualization tool for netlists `src/draw_netlist.py`.
4. Create abstractions for a chip.
5. Implement randomized iterative placement algorithm.
6. Implement simulated annealing placement algorithm.
7. Create abstractions for a plan (chip approximation for analytical placement).
8. Implement quadratic placement algorithms.
9. Create tool to visualize placement algorithms `src/plot_iterations.py`, `src/plot_snapshots.py`.
10. Write report.

Appendix A

Program output

```
Running demo...

Performing number of iterations experiment on iterative placement methods:
---------------------------------------------------------------------------
2000 atoms (1000 LUTs, 1000 FFs), 10 IPINs, 5 OPINs Netlist, (100, 100) Chip.
Random placement with 100 iterations. BBOX = 50093
Random placement with 1000 iterations. BBOX = 47260
Random placement with 10000 iterations. BBOX = 34095
Random placement with 100000 iterations. BBOX = 18442
Simulated annealing with 100 iterations. BBOX = 50093
Simulated annealing with 1000 iterations. BBOX = 46943
Simulated annealing with 10000 iterations. BBOX = 34277
Simulated annealing with 100000 iterations. BBOX = 18717

Performing number of recursions experiment on quadratic placement method:
---------------------------------------------------------------------------
2000 atoms (1000 LUTs, 1000 FFs), 10 IPINs, 5 OPINs Netlist, (100, 100) Chip.
Quadratic placement + adaptive partitioning with 1 recursions. BBOX = 36124
Quadratic placement + adaptive partitioning + 10000 iterations random placement with 1 recursions. BBOX = 25085
Quadratic placement + adaptive partitioning + 10000 iterations simulated annealing with 1 recursions. BBOX = 25600
Quadratic placement + adaptive partitioning with 2 recursions. BBOX = 34749
Quadratic placement + adaptive partitioning + 10000 iterations random placement with 2 recursions. BBOX = 24165
Quadratic placement + adaptive partitioning + 10000 iterations simulated annealing with 2 recursions. BBOX = 24134
Quadratic placement + adaptive partitioning with 3 recursions. BBOX = 36971
Quadratic placement + adaptive partitioning + 10000 iterations random placement with 3 recursions. BBOX = 25666
Quadratic placement + adaptive partitioning + 10000 iterations simulated annealing with 3 recursions. BBOX = 26049
Quadratic placement + adaptive partitioning with 4 recursions. BBOX = 42990
Quadratic placement + adaptive partitioning + 10000 iterations random placement with 4 recursions. BBOX = 28818
Quadratic placement + adaptive partitioning + 10000 iterations simulated annealing with 4 recursions. BBOX = 28946
Quadratic placement + bisection partitioning with 1 recursions. BBOX = 36124
Quadratic placement + bisection partitioning + 100000 iterations random placement with 1 recursions. BBOX = 25085
Quadratic placement + bisection partitioning + 100000 iterations simulated annealing with 1 recursions. BBOX = 25600
Quadratic placement + bisection partitioning with 2 recursions. BBOX = 36867
Quadratic placement + bisection partitioning + 100000 iterations random placement with 2 recursions. BBOX = 24882
Quadratic placement + bisection partitioning + 100000 iterations simulated annealing with 2 recursions. BBOX = 25297
Quadratic placement + bisection partitioning with 3 recursions. BBOX = 39774
Quadratic placement + bisection partitioning + 100000 iterations random placement with 3 recursions. BBOX = 27730
Quadratic placement + bisection partitioning + 100000 iterations simulated annealing with 3 recursions. BBOX = 27663
Quadratic placement + bisection partitioning with 4 recursions. BBOX = 49957
Quadratic placement + bisection partitioning + 100000 iterations random placement with 4 recursions. BBOX = 31985
Quadratic placement + bisection partitioning + 100000 iterations simulated annealing with 4 recursions. BBOX = 31870

Performing number of atoms experiment:
---------------------------------------------------------------------------
Random placement with 200 atoms (100 LUTs, 100 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 3595
Simulated annealing with 200 atoms (100 LUTs, 100 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 3441
Quadratic placement + adaptive partitioning with 200 atoms (100 LUTs, 100 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 3663
Quadratic placement + adaptive partitioning + 10000 iterations random placement with 200 atoms (100 LUTs, 100 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 2422
Quadratic placement + adaptive partitioning + 10000 iterations simulated annealing with 200 atoms (100 LUTs, 100 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 2500
Quadratic placement + bisection partitioning with 200 atoms (100 LUTs, 100 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 3919
Quadratic placement + bisection partitioning + 10000 iterations random placement with 200 atoms (100 LUTs, 100 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 2531
Quadratic placement + bisection partitioning + 10000 iterations simulated annealing with 200 atoms (100 LUTs, 100 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 2526
Random placement with 400 atoms (200 LUTs, 200 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 7844
Simulated annealing with 400 atoms (200 LUTs, 200 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 7919
Quadratic placement + adaptive partitioning with 400 atoms (200 LUTs, 200 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 8924
Quadratic placement + adaptive partitioning + 10000 iterations random placement with 400 atoms (200 LUTs, 200 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 4980
Quadratic placement + adaptive partitioning + 10000 iterations simulated annealing with 400 atoms (200 LUTs, 200 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 5040
Quadratic placement + bisection partitioning with 400 atoms (200 LUTs, 200 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 7974
Quadratic placement + bisection partitioning + 10000 iterations random placement with 400 atoms (200 LUTs, 200 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 4552
Quadratic placement + bisection partitioning + 10000 iterations simulated annealing with 400 atoms (200 LUTs, 200 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 4616
Random placement with 600 atoms (300 LUTs, 300 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 11785
Simulated annealing with 600 atoms (300 LUTs, 300 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 12247
Quadratic placement + adaptive partitioning with 600 atoms (300 LUTs, 300 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 10919
Quadratic placement + adaptive partitioning + 10000 iterations random placement with 600 atoms (300 LUTs, 300 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 6897
Quadratic placement + adaptive partitioning + 10000 iterations simulated annealing with 600 atoms (300 LUTs, 300 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 6433
Quadratic placement + bisection partitioning with 600 atoms (300 LUTs, 300 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 11752
Quadratic placement + bisection partitioning + 10000 iterations random placement with 600 atoms (300 LUTs, 300 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 6928
Quadratic placement + bisection partitioning + 10000 iterations simulated annealing with 600 atoms (300 LUTs, 300 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 7230
Random placement with 800 atoms (400 LUTs, 400 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 16373
Simulated annealing with 800 atoms (400 LUTs, 400 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 17164
Quadratic placement + adaptive partitioning with 800 atoms (400 LUTs, 400 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 13059
Quadratic placement + adaptive partitioning + 10000 iterations random placement with 800 atoms (400 LUTs, 400 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 7929
Quadratic placement + adaptive partitioning + 10000 iterations simulated annealing with 800 atoms (400 LUTs, 400 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 8395
Quadratic placement + bisection partitioning with 800 atoms (400 LUTs, 400 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 13573
Quadratic placement + bisection partitioning + 10000 iterations random placement with 800 atoms (400 LUTs, 400 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 8077
Quadratic placement + bisection partitioning + 10000 iterations simulated annealing with 800 atoms (400 LUTs, 400 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 7995
Random placement with 1000 atoms (500 LUTs, 500 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 20850
Simulated annealing with 1000 atoms (500 LUTs, 500 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 20650
Quadratic placement + adaptive partitioning with 1000 atoms (500 LUTs, 500 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 17732
Quadratic placement + adaptive partitioning + 10000 iterations random placement with 1000 atoms (500 LUTs, 500 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 11659
Quadratic placement + adaptive partitioning + 10000 iterations simulated annealing with 1000 atoms (500 LUTs, 500 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 11385
Quadratic placement + bisection partitioning with 1000 atoms (500 LUTs, 500 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 18235
Quadratic placement + bisection partitioning + 10000 iterations random placement with 1000 atoms (500 LUTs, 500 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 10850
Quadratic placement + bisection partitioning + 10000 iterations simulated annealing with 1000 atoms (500 LUTs, 500 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 10735
Random placement with 1200 atoms (600 LUTs, 600 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 26943
Simulated annealing with 1200 atoms (600 LUTs, 600 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 27241
Quadratic placement + adaptive partitioning with 1200 atoms (600 LUTs, 600 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 20127
Quadratic placement + adaptive partitioning + 10000 iterations random placement with 1200 atoms (600 LUTs, 600 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 13942
Quadratic placement + adaptive partitioning + 10000 iterations simulated annealing with 1200 atoms (600 LUTs, 600 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 13955
Quadratic placement + bisection partitioning with 1200 atoms (600 LUTs, 600 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 24178
Quadratic placement + bisection partitioning + 10000 iterations random placement with 1200 atoms (600 LUTs, 600 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 15649
Quadratic placement + bisection partitioning + 10000 iterations simulated annealing with 1200 atoms (600 LUTs, 600 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 15241
```

```
Random placement with 1400 atoms (700 LUTs, 700 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 34419
Simulated annealing with 1400 atoms (700 LUTs, 700 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 34716
Quadratic placement + adaptive partitioning with 1400 atoms (700 LUTs, 700 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 24565
Quadratic placement + adaptive partitioning + 10000 iterations random placement with 1400 atoms (700 LUTs, 700 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 17445
Quadratic placement + adaptive partitioning + 10000 iterations simulated annealing with 1400 atoms (700 LUTs, 700 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 17614
Quadratic placement + bisection partitioning with 1400 atoms (700 LUTs, 700 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 25342
Quadratic placement + bisection partitioning + 10000 iterations random placement with 1400 atoms (700 LUTs, 700 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 17412
Quadratic placement + bisection partitioning + 10000 iterations simulated annealing with 1400 atoms (700 LUTs, 700 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 17419
Random placement with 1600 atoms (800 LUTs, 800 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 39017
Simulated annealing with 1600 atoms (800 LUTs, 800 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 39704
Quadratic placement + adaptive partitioning with 1600 atoms (800 LUTs, 800 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 26720
Quadratic placement + adaptive partitioning + 10000 iterations random placement with 1600 atoms (800 LUTs, 800 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 19288
Quadratic placement + adaptive partitioning + 10000 iterations simulated annealing with 1600 atoms (800 LUTs, 800 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 19458
Quadratic placement + bisection partitioning with 1600 atoms (800 LUTs, 800 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 27412
Quadratic placement + bisection partitioning + 10000 iterations random placement with 1600 atoms (800 LUTs, 800 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 18988
Quadratic placement + bisection partitioning + 10000 iterations simulated annealing with 1600 atoms (800 LUTs, 800 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 18447
Random placement with 1800 atoms (900 LUTs, 900 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 48008
Simulated annealing with 1800 atoms (900 LUTs, 900 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 47633
Quadratic placement + adaptive partitioning with 1800 atoms (900 LUTs, 900 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 35858
Quadratic placement + adaptive partitioning + 10000 iterations random placement with 1800 atoms (900 LUTs, 900 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 26549
Quadratic placement + adaptive partitioning + 10000 iterations simulated annealing with 1800 atoms (900 LUTs, 900 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 26113
Quadratic placement + bisection partitioning with 1800 atoms (900 LUTs, 900 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 39565
Quadratic placement + bisection partitioning + 10000 iterations random placement with 1800 atoms (900 LUTs, 900 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 27540
Quadratic placement + bisection partitioning + 10000 iterations simulated annealing with 1800 atoms (900 LUTs, 900 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 27847
Random placement with 2000 atoms (1000 LUTs, 1000 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 50337
Simulated annealing with 2000 atoms (1000 LUTs, 1000 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 50637
Quadratic placement + adaptive partitioning with 2000 atoms (1000 LUTs, 1000 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 40603
Quadratic placement + adaptive partitioning + 10000 iterations random placement with 2000 atoms (1000 LUTs, 1000 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 28791
Quadratic placement + adaptive partitioning + 10000 iterations simulated annealing with 2000 atoms (1000 LUTs, 1000 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 28732
Quadratic placement + bisection partitioning with 2000 atoms (1000 LUTs, 1000 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 43347
Quadratic placement + bisection partitioning + 10000 iterations random placement with 2000 atoms (1000 LUTs, 1000 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 30126
Quadratic placement + bisection partitioning + 10000 iterations simulated annealing with 2000 atoms (1000 LUTs, 1000 FFs), 10 IPINs, 5 OPINs netlist. BBOX = 30603

Performing number of phases experiment:
----------------------------------------------------------------------------
Random placement with 1 phases. BBOX = 36494
Simulated annealing with 1 phases. BBOX = 35680
Quadratic placement + adaptive partitioning with 1 phases. BBOX = 36976
Quadratic placement + adaptive partitioning + 10000 iterations random placement with 1 phases. BBOX = 25398
Quadratic placement + adaptive partitioning + 10000 iterations simulated annealing with 1 phases. BBOX = 25296
Quadratic placement + bisection partitioning with 1 phases. BBOX = 48956
Quadratic placement + bisection partitioning + 10000 iterations random placement with 1 phases. BBOX = 30079
Quadratic placement + bisection partitioning + 10000 iterations simulated annealing with 1 phases. BBOX = 30567
Random placement with 2 phases. BBOX = 35428
Simulated annealing with 2 phases. BBOX = 35411
Quadratic placement + adaptive partitioning with 2 phases. BBOX = 40521
Quadratic placement + adaptive partitioning + 10000 iterations random placement with 2 phases. BBOX = 28916
Quadratic placement + adaptive partitioning + 10000 iterations simulated annealing with 2 phases. BBOX = 28209
Quadratic placement + bisection partitioning with 2 phases. BBOX = 40440
Quadratic placement + bisection partitioning + 10000 iterations random placement with 2 phases. BBOX = 26385
Quadratic placement + bisection partitioning + 10000 iterations simulated annealing with 2 phases. BBOX = 26967
Random placement with 3 phases. BBOX = 34095
Simulated annealing with 3 phases. BBOX = 34277
Quadratic placement + adaptive partitioning with 3 phases. BBOX = 34749
Quadratic placement + adaptive partitioning + 10000 iterations random placement with 3 phases. BBOX = 24165
Quadratic placement + adaptive partitioning + 10000 iterations simulated annealing with 3 phases. BBOX = 24134
Quadratic placement + bisection partitioning with 3 phases. BBOX = 36867
Quadratic placement + bisection partitioning + 10000 iterations random placement with 3 phases. BBOX = 24882
Quadratic placement + bisection partitioning + 10000 iterations simulated annealing with 3 phases. BBOX = 25297
Random placement with 4 phases. BBOX = 36118
Simulated annealing with 4 phases. BBOX = 35557
Quadratic placement + adaptive partitioning with 4 phases. BBOX = 42573
Quadratic placement + adaptive partitioning + 10000 iterations random placement with 4 phases. BBOX = 30270
Quadratic placement + adaptive partitioning + 10000 iterations simulated annealing with 4 phases. BBOX = 30567
Quadratic placement + bisection partitioning with 4 phases. BBOX = 43640
Quadratic placement + bisection partitioning + 10000 iterations random placement with 4 phases. BBOX = 30982
Quadratic placement + bisection partitioning + 10000 iterations simulated annealing with 4 phases. BBOX = 30908
Random placement with 5 phases. BBOX = 36825
Simulated annealing with 5 phases. BBOX = 37333
Quadratic placement + adaptive partitioning with 5 phases. BBOX = 36362
Quadratic placement + adaptive partitioning + 10000 iterations random placement with 5 phases. BBOX = 26666
Quadratic placement + adaptive partitioning + 10000 iterations simulated annealing with 5 phases. BBOX = 26687
Quadratic placement + bisection partitioning with 5 phases. BBOX = 38331
Quadratic placement + bisection partitioning + 10000 iterations random placement with 5 phases. BBOX = 26993
Quadratic placement + bisection partitioning + 10000 iterations simulated annealing with 5 phases. BBOX = 27406
Random placement with 6 phases. BBOX = 35333
Simulated annealing with 6 phases. BBOX = 35066
Quadratic placement + adaptive partitioning with 6 phases. BBOX = 49786
Quadratic placement + adaptive partitioning + 10000 iterations random placement with 6 phases. BBOX = 34931
Quadratic placement + adaptive partitioning + 10000 iterations simulated annealing with 6 phases. BBOX = 34655
Quadratic placement + bisection partitioning with 6 phases. BBOX = 48300
Quadratic placement + bisection partitioning + 10000 iterations random placement with 6 phases. BBOX = 33623
Quadratic placement + bisection partitioning + 10000 iterations simulated annealing with 6 phases. BBOX = 34035
Random placement with 7 phases. BBOX = 38671
Simulated annealing with 7 phases. BBOX = 39112
Quadratic placement + adaptive partitioning with 7 phases. BBOX = 52269
Quadratic placement + adaptive partitioning + 10000 iterations random placement with 7 phases. BBOX = 36009
Quadratic placement + adaptive partitioning + 10000 iterations simulated annealing with 7 phases. BBOX = 36649
Quadratic placement + bisection partitioning with 7 phases. BBOX = 51808
Quadratic placement + bisection partitioning + 10000 iterations random placement with 7 phases. BBOX = 36288
Quadratic placement + bisection partitioning + 10000 iterations simulated annealing with 7 phases. BBOX = 36299
Random placement with 8 phases. BBOX = 36629
Simulated annealing with 8 phases. BBOX = 36959
Quadratic placement + adaptive partitioning with 8 phases. BBOX = 52803
Quadratic placement + adaptive partitioning + 10000 iterations random placement with 8 phases. BBOX = 36124
Quadratic placement + adaptive partitioning + 10000 iterations simulated annealing with 8 phases. BBOX = 36763
Quadratic placement + bisection partitioning with 8 phases. BBOX = 52912
Quadratic placement + bisection partitioning + 10000 iterations random placement with 8 phases. BBOX = 36231
```

```
Quadratic placement + bisection partitioning + 10000 iterations simulated annealing with 8 phases. BBOX = 36736
Random placement with 9 phases. BBOX = 39544
Simulated annealing with 9 phases. BBOX = 40710
Quadratic placement + adaptive partitioning with 9 phases. BBOX = 55697
Quadratic placement + adaptive partitioning + 10000 iterations random placement with 9 phases. BBOX = 39508
Quadratic placement + adaptive partitioning + 10000 iterations simulated annealing with 9 phases. BBOX = 39400
Quadratic placement + bisection partitioning with 9 phases. BBOX = 55060
Quadratic placement + bisection partitioning + 10000 iterations random placement with 9 phases. BBOX = 38572
Quadratic placement + bisection partitioning + 10000 iterations simulated annealing with 9 phases. BBOX = 39626
Random placement with 10 phases. BBOX = 44157
Simulated annealing with 10 phases. BBOX = 44885
Quadratic placement + adaptive partitioning with 10 phases. BBOX = 63564
Quadratic placement + adaptive partitioning + 10000 iterations random placement with 10 phases. BBOX = 44446
Quadratic placement + adaptive partitioning + 10000 iterations simulated annealing with 10 phases. BBOX = 44146
Quadratic placement + bisection partitioning with 10 phases. BBOX = 64059
Quadratic placement + bisection partitioning + 10000 iterations random placement with 10 phases. BBOX = 44105
Quadratic placement + bisection partitioning + 10000 iterations simulated annealing with 10 phases. BBOX = 44594
```

Resources

1. M. Sarrafzadeh, M. Wang, X. Yang. *Modern Placement Techniques.* Springer, 2002.
2. R. Rutenbar. "VLSI CAD Part II: Layout." Internet : `https://www.coursera.org/learn/vlsi-cad-layout`, [Mar. 19, 2018]
3. J. Nocedal, S. Wright. *Numerical Optimization*, 2nd Edition. Springer, 2006.

References

[1] "Carry-Lookahead Adder." Internet: `https://www.electronicshub.org/carry-look-ahead-adder/`, Jun. 29, 2015 [Mar. 25, 2018]

[2] "Dual Clock FIFO." Internet : `http://web.ece.ucdavis.edu/~astill/dcfifo.html`, [Mar. 25, 2018]

[3] "FPGA Architecture." Internet : `https://www.altera.com/en_US/pdfs/literature/wp/wp-01003.pdf`, Jul., 2006 [Mar. 25, 2018]

[4] "Partially Reconfiguring a Design on Intel Arria 10 GX FPGA Development Board." Internet : `https://www.altera.com/documentation/ihj1482170009390.html`, Nov. 06, 2017 [Mar. 25, 2018]

[5] "Boost C++ Libraries." Internet: `https://www.boost.org/`, [Apr. 15, 2018]

[6] "Eigen." Internet: `http://eigen.tuxfamily.org`, [Apr. 15, 2018]

[7] "Robust Cholesky decomposition of a matrix with pivoting." Internet: `https://eigen.tuxfamily.org/dox/classEigen_1_1LDLT.html`, [Apr. 15, 2018]