Contents

Preface

Since the beginning of early 50's circuit manufacturing with large number of components has began. Today VLSI design is enriched with many sophisticated algorithms and development of Physical Design Automation tools. Many problems is this field have been classified as NP-Complete, thus VLSI has been an popular research domain in recent years.

This document deals with the placement problem from the physical design process. It attempts to deliver a feel of the recent trends in the placement techniques. The document is actually a documentation of our class assignment for M.Tech. 2nd semester course.

Errors and Omission

Despite of our best effort, this document may contain some errors. If you find any error or have any constructive suggestion, we would certainly appreciate your comments. You can email us at **ratcoinc@gmail.com** or **arjaitapal25@gmail.com**. Unfortunately we could not have implemented any detailed placement technique. We plan to extend this work in future.

Acknowledgments

Firstly we would like to thank Prof. Pritha Banerjee for wonderful lectures on VLSI. She also taught us some of the advanced and recent trends in VLSI field. We would like to thank to Sudipta Paul. Also we would like to state that, this document is fully written in MiKTEX, an implementation of LATEX for Windows.

We thank again Prof. Pritha Banerjee for giving us this assignment work, which greatly improved our insight into the modern placement algorithms for VLSI. Also we have learnt many new features available in the Apache Commons Mathematics Library.

15th August, 2016

Rathindra Nath Dutta & Arjaita Pal
Master of Technology
Department of Computer Science and Engineering
Calcutta University

Introduction

The IC technology has evolved rapidly over the past few years due to development of efficient algorithms. The Computer Aided Design (CAD) tools has made most of the design phases partially or fully automated. The VLSI design cycle starts with formal specification of a VLSI chip, follows a series of steps, and eventually produces a packaged chip. A typical design cycle may be represented by the flowchart shown in figure 1.1 [1][2].

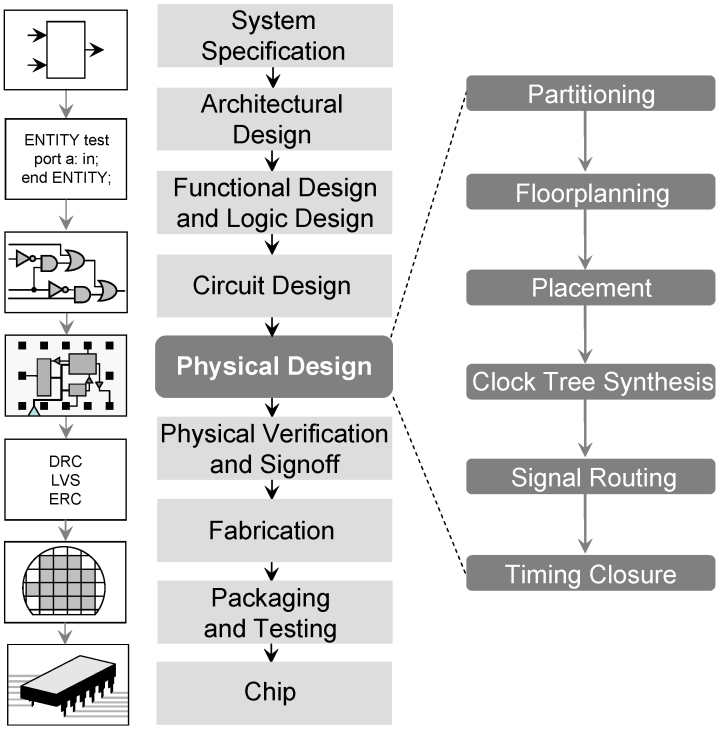

Figure 1.1: VLSI design flow

In this document our main concern is the physical design phase, and more specifically the placement techniques. Given a circuit design (i.e. netlist), it is converted into a geometric representation called *layout*. The exact details of a layout depends on the *design rules* of the underlying fabrication process.

The physical design process starts with *partitioning* the circuit into smaller manageable sub-circuits, called blocks. The partitioning is generally done in hierarchical manner. Then comes *floorplanning*, where a good layout is selected from

multiple alternatives for each blocks. During *placement*, the blocks are exactly placed on the chip. Then *routing* is done. The objective of routinng is to plan the interconnection paths between blocks according to the netlist.

1.1 Placement

Placement is the process of determining the locations of circuit devices on a die surface. It is an important stage in the VLSI design flow, because it affects routability, performance, heat distribution, and to a less extent, power consumption of a design. Traditionally, it is applied after the logic synthesis stage and before the routing stage [3].

Placement is a computationally difficult problem. Even the simple case of placing a circuit with only unit-size modules and 2-pin nets along a straight line to minimize total wirelength is *NP-complete* [3]. The real life placement problem is much more complicated. The circuit may contain modules of different sizes and may have multi-pin nets. The placement region is two-dimensional, most popular cost function is based on total wirelength, though other cost functions are used. As designs with millions of modules are now common, it is a major challenge to design efficient placement algorithms to produce high-quality placement solutions.

One way to overcome this complexity issue is to perform placement in multiple steps. **Global placement** aims at generating a rough placement solution that may violate some placement constraints (e.g. overlaps among modules) while maintaining a global view of the whole netlist. **Legalization** makes the rough solution from global placement legal (i.e., no placement constraint violation) by moving modules around locally. **Detailed placement** further improves the legalized placement solution in an iterative manner by rearranging a small group of modules in a local region while keeping all other modules fixed.

The global placement step is the most crucial one of the three as it has the most impact on placement solution quality and runtime. Thus it has been the focus of most research works. The most commonly used global placement approaches are partitioning-based approach, simulated annealing approach, and analytical approach, of which the analytical approach is currently the best option in both quality and runtime. The global placement is often done in two phases. In the first phase an initial placement solution is obtained, and in the second phase the solution is improved by iterative refinements.

1.2 Placement problem

The input to the placement problem is a placement region, a set of modules, and a set of nets. The widths and heights of the placement region and all modules are already determined in the floorplanning step. The locations of I/O pins on the placement region and on all modules are fixed. Sometimes, some input modules (e.g., buffer bays, I/O modules, IP blocks) are preplaced by designers, and, hence, their locations are also fixed before placement. Each net specifies a collection of

pins in the placement region and/or in some modules that are connected. Basically, placement is to find a position for each module within the placement region so that there is no overlap among the modules and some objective is optimized.

Many variations in the placement problem formulation exist, because different designs may require different objectives and different design styles may introduce different constraints. In a **standard-cell design**, all modules have the same height. The placement of standard cells has to be aligned with some prespecified standard-cell rows in the placement region. Because of the popularity of standard-cell design, most placement algorithms assume a standard-cell design style.

Total wirelength is the most commonly used objective in placement formulations. Minimization of total wirelength indirectly optimizes several other objectives [3]. First, *routability* can be improved by less routing demand. Second, *timing* can be better because shorter wires have less delay. Third, *power consumption* can be reduced because shorter wires also introduce less capacitive load. To specify the importance of different nets in optimizing another objective, a weight can be assigned to each net. Then the total weighted wirelength will be a much better objective. It is difficult to predict during placement the actual wirelength of a net after routing, because it is router-dependent. Several approaches are used to estimate the routed wirelength for a given placement. The most widely used approach is half-perimeter wirelength (**HPWL**). The HPWL of a net is equal to half of the perimeter of the smallest bounding rectangle that encloses all the pins of the net. For example, for the 5-pin net in Figure **??**a, the HPWL is W + H as shown in Figure 1.2b.

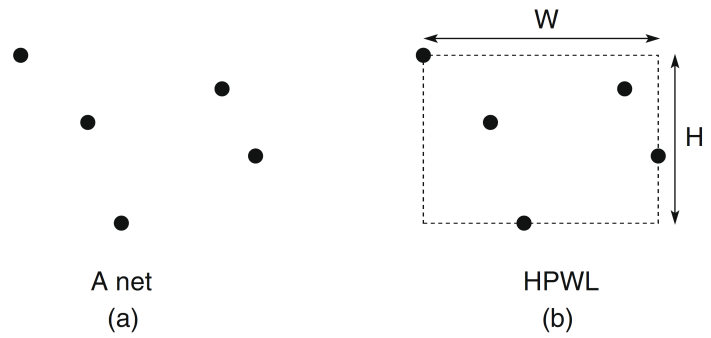

Figure 1.2: Wirelength estimation

HPWL is popular because it can be computed in linear time and it can be written as a simple function of the coordinates of the pins (described in section ***). It also provides exact wirelength for optimally routed nets with two and three pins. However, HPWL can significantly underestimate the wirelength for nets with four or more pins. Alternate approaches of wirelength estimation includes based rectilinear minimum spanning tree (RMST) and rectilinear Steiner minimal tree (RSMT).

Chapter 2 of this document is fully devoted to analytical approach for global placement. First we will discuss the *Quadratic technique* to obtain an initial place-

ment solution. Then we will discuss a non-linear analytical model for placement, which involves a newer wirelength estimation model called *Log-Sum-Exponentiation* (LSE). Chapter 3 describes the legalization technique we have used. The details about our implementation of the these techniques are presented in chapter 4.

Analytical Placement

The analytical approach for global placement tries to express the cost function as an objective and the constraints are given as analytical functions involving coordinates of the modules. Analytical approach is very much effective and popular for standard cell design.

The following notations are used through out this document. In a placement problem we have a set of modules V, which are considered as vertices and a set of nets E, considered as edges. The modules in V are indexed from 1 to n. Each module i has its own width, height, x and y-coordinates, which are denoted as w_i, h_i, x_i, and y_i. Each net $e \in E$ is a set of modules which are to be made electrically equivalent. As stated earlier in chapter 1, some modules have fixed location for which the x and y-coordinates are constants, otherwise these are variables. The width and height of a module is constant in all cases and generally determined in floorplanning phase. A net is connected to a specific point on a module, called pin. For simplicity we are assuming that all pins are located at the center of the module.

2.1 Problem formulation

In this section our goal is to formulate a mathematical program for a given placement problem. Such an optimization problem have an objective function to be optimized and a set of constraints written as mathematical functions. Let us consider the simple HPWL wirelength model for our objective function.

The HPWL of net $e \in E$ can be written as:

$$HPWL_e(x_1, ..., x_n, y_1, ..., y_n) = \left(\max_{i \in e}\{x_i\} - \min_{i \in e}\{x_i\}\right) + \left(\max_{i \in e}\{y_i\} - \min_{i \in e}\{y_i\}\right)$$

To specify the non-overlapping constraint we define:

$$Overlap_{ij}(x_i, y_i, x_j, y_j) = \Theta\left(\left[x_i - \tfrac{w_i}{2}, x_i + \tfrac{w_i}{2}\right], \left[x_j - \tfrac{w_j}{2}, x_j + \tfrac{w_j}{2}\right]\right) \times \Theta\left(\left[y_i - \tfrac{h_i}{2}, y_i + \tfrac{h_i}{2}\right], \left[y_j - \tfrac{h_j}{2}, y_j + \tfrac{h_j}{2}\right]\right)$$

The function Θ is defined as:

$$\Theta([L1, R1], [L2, R2]) = [min(R1, R2) - max(L1, L2)]^+$$

where

$$[z]^+ = \begin{cases} z, & if\ z > 0 \\ 0, & if\ z \leq 0 \end{cases}$$

The function $\Theta([L1, R1], [L2, R2])$ gives the length of the overlapping region of the interval $[L1, R1]$ and $[L2, R2]$ as illustrated in figure 2.1[3].

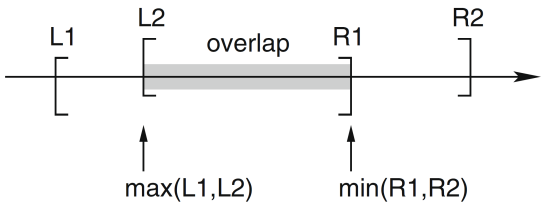

Figure 2.1: The overlapping region

Now, the placement problem can be given as:

$$\begin{array}{ll} \text{Minimize} & \sum_{e \in E} c_e \times HPWL_e(x_1, ..., x_n, y_1, ..., y_n) \\ \text{Subject to} & Overlap_{ij}(x_i, y_i, x_j, y_j) = 0, \text{ for all } i, j \in V \text{ such that } i \neq j \end{array}$$

Here c_e are constants and are used to give priority to nets. Additional constraints may also be added to this formulation. As it is evident that both our objective function and the overlap constraint function are non-differentiable. Hence this seemingly simple problem becomes extremely difficult to handle. Moreover, there are $\frac{n(n-1)}{2}$ constraints. In real life placement problems the n is in the order of millions, thus above formulation is not a feasible one. Hence the functions are replaced by some differentiable functions as shown in the following sections.

2.2 Quadratic programming

The simple solution to the above problem is to replace the objective function i.e. HPWL by a quadratic wirelength function. For 2-pin net HPWL for the net is essentially the Manhattan distance between the modules i and j, i.e.,

$$HPWL_{i,j} = |x_i - x_j| + |y_i - y_j|$$

The **quadratic wirelength** is defined as the squared Euclidean distance between the modules i and j, i.e.,

$$QWL_{i,j} = (x_i - x_j)^2 + (y_i - y_j)^2$$

Thus our cost function becomes: $L = \frac{1}{2} \sum_{1 \leq i < j \leq n} c_{i,j} \times (x_i - x_j)^2 + (y_i - y_j)^2$

The above function is both convex and differentiable. The half is taken of the total wirelength in order to have the derivatives in simpler form. This equation can be transformed into matrix form:

$$L = \tfrac{1}{2}\mathbf{x^T Q x} + \mathbf{d_x^T x} + \mathbf{y^T Q y} + \mathbf{d_y^T y} + \text{constant terms}$$

Assume that modules 1 to r are movable and modules $r+1$ to n are fixed ones. Therefore $x_1, ..., x_r$ and $y_1, ..., y_r$ are the only variables while $x_{r+1}, ..., x_n$ and $y_{r+1}, ..., y_n$ are constants. We define $\mathbf{x}$ and $\mathbf{y}$ as column matrices as follows:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_r \end{bmatrix}, \ \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_r \end{bmatrix}$$

Now we define connectivity matrix $C = (c_{i,j})_{r\times r}$, where $c_{i,j}$ denotes the connectivity between modules i and j. We also define a diagonal matrix $D = (d_{i,j})_{r\times r}$ such that $d_{i,i} = \sum_{j\in V} c_{i,j}$ Now Q is defined as $Q = D - C$. Another column matrix d_x is defined as:

$$d_x = \begin{bmatrix} d_{x_1} \\ d_{x_2} \\ \vdots \\ d_{x_r} \end{bmatrix}, \ where\ d_{x_i} = -\sum_{j\in\{r+1,...,n\}} c_{i,j}x_j$$

Similarly d_y is defined. This can be illustrated using an example. Consider a circuit with 3 movable modules and 3 fixed modules. The aforesaid matrices can be obtained as shown in figure 2.2[3]. Since our cost function L is convex and

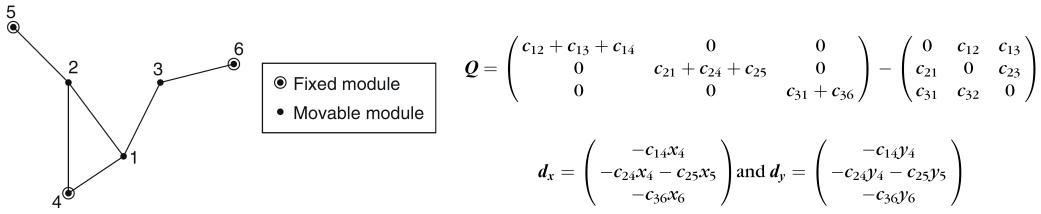

Figure 2.2: Example of quadratic formulation

differentiable it can easily minimized. As we know the minima can exist when first order derivative is zero. Now $\frac{\delta L}{\delta x} = Qx + d_x$ and $\frac{\delta L}{\delta y} = Qy + d_y$. Therefore wirelength will be minimum when $Qx + d_x = 0$ and $Qy + d_y = 0$. Thus the problem reduces to solving these equation which represents a system of linear equations.

If all movable modules are connected to fixed modules either directly or indirectly, Q is positive definite and thus invertible. This implies the existence of a unique global optimal solution. The simplicity of quadratic formulation is the main reason for its popularity.

2.2.1 Handling multi-pin nets

The traditional way to handle multi-pin nets is to replace such a net with a clique (or complete graph). A newer technique is often used called star model, where a

new (dummy) dimensionless module is introduced for a multi-pin nets and all the modules in that net are connected to that new module, thus forming a star like structure. These are illustrated in figure 2.3.

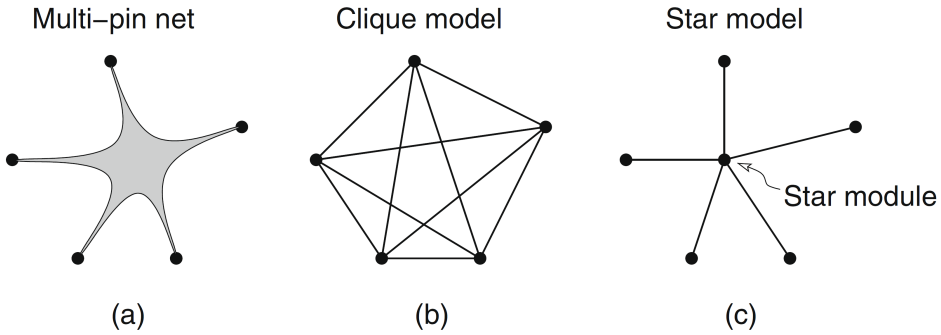

Figure 2.3: Handling multi-pin nets

2.2.2 Handling non-overlapping constraint

The above formulation does not consider the overlapping of the modules. Hence the solution of the above will try to place connected modules as close as possible and possibly one above another. This clearly emphasize the fact that minimizing the wirelength and to avoid module overlap are two conflicting goals. Hence in practice a quadratic formulation is solved without considering the overlaps. Once the result is obtained iterative local refinement is applied to spread the modules.

2.3 Nonlinear programming

Another way to use analytical placement is to replace the non-differentiable objective function and non-overlapping constraints by some nonlinear function which is convex and easy to differentiate. In this model we use approximation instead of exact values of these functions. In other words the nonlinear function is approximation to the functions defined section 2.1. This approach is used in many modern placers, namely APlace, NTUPlace, mPL etc.[3]. These algorithms use a new wirelength model (described below), and use density based non-overlapping constraint proposed by Naylor et. al. [4].

2.3.1 Log-Sum-Exponentiation wirelength model

The Log-Sum-Exponentiation (or LSE in short) is a multivariate function defined as $LSE_\alpha(z_1, \ldots, z_n) = \alpha \times \left(\log\left(\sum_{i=1}^{n} e^{z_i/\alpha}\right)\right)$. This is an approximation to the maximum function, and α is a parameter which controls the accuracy of the approximation. Thus we have $\lim_{\alpha \to 0} LSE_\alpha(z_1, \ldots, z_n) = \max(z_1, \ldots, z_n)$. In section

2.1, the HPWL is defined as a combination of both maximum and minimum function, which can be written only in terms of maximum functions as follows:

$$\begin{aligned} &HPWL_e(x_1, ..., \ x_n, \ y_1, ..., \ y_n) \\ &= \left(\max_{i \in e}\{x_i\} - \min_{i \in e}\{x_i\}\right) + \left(\max_{i \in e}\{y_i\} - \min_{i \in e}\{y_i\}\right) \\ &= \left(\max_{i \in e}\{x_i\} + \max_{i \in e}\{-x_i\}\right) + \left(\max_{i \in e}\{y_i\} + \max_{i \in e}\{-y_i\}\right) \end{aligned}$$

So, HPWL can be approximated by the LSE wirelength as follows:

$$\begin{aligned} &LSEWL_{e,\alpha}(x_1, ..., \ x_n, \ y_1, ..., \ y_n) \\ &= \alpha \times \left(\log\left(\sum_{i=1}^{n} e^{x_i/\alpha}\right) + \log\left(\sum_{i=1}^{n} e^{-x_i/\alpha}\right) + \log\left(\sum_{i=1}^{n} e^{y_i/\alpha}\right) + \log\left(\sum_{i=1}^{n} e^{-y_i/\alpha}\right)\right) \end{aligned}$$

The $LSEWL_{e,\alpha}()$ is strictly convex, continuously differentiable , and converges to $HPWL_e()$ as α converges to zero[4].

2.3.2 Non-overlapping constraints

Though the density based constraints are popular to handle non-overlapping constraints [3], we have used a simpler and easy to implement formulation. Recall that we have defined Overlap function in terms of another function $\theta()$ in section 2.1. The main problem with this function is that it is not differentiable. So we convert it to a differentiable one as follows:

$$\begin{aligned} \Theta([L1, R1], [L2, R2]) &= [min(R1, R2) - max(L1, L2)]^+ \\ &= [-max(-R1, -R2) - max(L1, L2)]^+ \\ &\approx [-LSE_\alpha(-R1, -R2) - LSE_\alpha(L1, L2)]^+ \end{aligned}$$

Now we redefine the function $[z]^+$ such that the overall function is smooth along with continuous and differentiable. One simple solution may be: $[z]^+ = e^{z/\alpha} - 1$

2.3.3 The nonlinear program

We have defined the mathematical program for the placement problem in section 2.1. That formulation had a objective function (wirelength) to be minimized with non-overlapping constraints. Let us convert that to a unconstrained minimization problem as follows:

Minimize $\sum_{e \in E} c_e \times LSEWL_{e,\alpha}(x_1, ..., \ x_n, \ y_1, ..., \ y_n) + \beta \times \sum_{\substack{i,j \in V \\ i \neq j}} Overlap_{ij}(x_i, \ y_i, \ x_j, \ y_j)$

Here β is a parameter to specify the importance of the non-overlap constraint, and it is tuned iteratively. Such an unconstrained problem can be solved by the **conjugate gradient method**. Outline of the algorithm is presented in figure 2.4 [5].

Conjugate Gradient Algorithm

Input:

A high dimensional function $f(x)$
Initial solution x_0
Minimum step length ε
Initial maximum step length γ_0
Maximum number of iterations N

Output:

Local minimum x^*

Algorithm:

01. Initialize # iterations $k = 1$, step length $\alpha_0 = \infty$
gradients $g_0 = 0$ and conjugate directions $d_0 = 0$
02. **For** ($k < N$ and step length $\alpha_{k-1} > \varepsilon$)
03. Compute gradients $g_k = \nabla f(x_k)$
04. Compute Polak-Ribiere parameter $\beta_k = \frac{g_k^T (g_k - g_{k-1})}{||g_{k-1}||^2}$
05. Compute conjugate directions $d_k = -g_k + \beta_k d_{k-1}$
06. Compute step length α_k within γ_{k-1}
using Golden Section line search algorithm
07. Update new solution $x_k = x_{k-1} + \alpha_k d_k$
08. Update maximum step length $\gamma_k = \text{MAX}\{\gamma_0, 2\gamma_{k-1}\}$
09. Return minimum $x^* = x_k$

Figure 2.4: Conjugate Gradient Algorithm

Legalization

Given an illegal placement solution, the legalization process eliminates all overlaps by perturbing the modules as little as possible. The analytical placement can not always remove all the overlaps, thus Legalization is a necessary step. For standard cell placement, it is easy to legalize the placement. The Tetris algorithm[6] is simple and yet popular for standard cell placement.

3.1 Tetris algorithm

For each module i ($1 \leq i \leq n$), we have its coordinate ($x_i,\ y_i$); the module must be places into one of the standard cell rows $R = \{r_1,\ r_2,\ \ldots,\ r_k\}$, and assume that leftmost open position in each row is known. Here the modules are first sorted in ascending order with respect to their x-coordinate. Then the modules are put one by one into left side of the row that minimizes the total displacement fo that module. The outline of this algorithm is as follows:

Algorithm 1 Tetris legalizer

```
Let M be the set of all modules(moveable) that need to legalized.
Sort the modules in M by their x-coordinates into a L_s.
Let l_j be the leftmost free position of each row r_j (1 ≤ j ≤ k)
for each module i (1 ≤ i ≤ n) do
    best = ∞
    for each row j (1 ≤ j ≤ k) do
        cost = displacement of moving module i in L_s to l_j.
        if cost ≤ best then
            best = cost
            best_row = j
        end if
    end for
    Move module i in L_s to row best_row.
    l_best_row = l_best_row + width_i
end for
```

If we rotate the placement region 90 deg counter-clockwise, the legalization process will look very much like a game of Tetris, for which the algorithm got its name. This simple greedy approach is extremely fast, which is the main reason

for its popularity. However, it may sometimes result into very uneven row length hence fail to produce good packing. Many improvement of this basic algorithm are proposed to overcome these problem.

Once the legalization is done Detailed Placement is applied. Domino, FastDP are some popular algorithms for detailed placement.

Implementation details

This chapter describes how the techniques and algorithms, mentioned in previous chapters, are implemented. We have used *Java programming language* (J2SE8) for our implementation as it is free, platform independent, portable, and most importantly has rich library support. The program first reads the input, i.e, a given placement problem instance from a set of files. It then generates a initial solution by quadratic technique. The obtained solution is then iteratively improved by the nonlinear programming using log-sum-exponentiation wirelength model. Finally the solution is legalized using Tetris algorithm. A graphical interface is also implemented for visualization.

4.1 Reading the input files

A good *placer* should read the data from files and also put solution into files, in this way it will be pluggable with any other programs. Here the Input data are to be read from some given files, which have some predefined formats. We are using the **Bookshelf format**[7][8] which are common for benchmark data and have been used in VLSI CAD contests like ISPD, ICCAD etc. The input is split into six different files all of which follows the following conventions: Blank characters are spaces and tabs and multiple blank characters are equivalent to a single one. The # is used for comment. All characters after # can be safely ignored by the file parser.

AUX file

To assemble multiple component files into a single problem instance, a .aux file is used. It typically include one line as shown below.

```
RowBasedPlacement: circuit.nodes circuit.nets circuit.wts\\
                                circuit.pl circuit.scl circuit.shapes
```

Specified files are parsed in a predefined order that is determined by their extensions (not the order in .aux file) specifically they should be parsed in the following order: .nodes, .nets, .wts, .scl, .pl.

NODES file

It gives details about the nodes (modules/cells/objects) present in the design. After the standard header, the format specifies *NumNodes* the total number of nodes and *NumTerminals* the number of terminal nodes. Then for each node a line is given specifying its *name*, *width*, *height*, and *movetype*. A node can be one of the three types either it is *moveable*, or it is fixed called *terminal*; among the fixed nodes some node allow overlap, they are called *terminal_NI*. If a line does not specify a movetype, the associated node is a movable node. A sample .nodes file is presented below.

```
# UCLA nodes 1.0
# File header with version information, etc.,
# Anything following # is a comment, and should be ignored

NumNodes        : 5
NumTerminals    : 2

# node-name  width  height terminal/movable

o0        4        9                    # movable node
o1        4        9
o2       24        9
o3      414     2007    terminal        # fixed node
p0        1        1    terminal_NI     # fixed node,\\
                        but overlap is allowed with this node
```

As it is evident that in this example we have $5 - 2 = 3$ moveable nodes, and all moveable nodes have same height since we are using standard cell design.

NETS file

It specifies the circuit net-list i.e., the set of nets or connections in the hypergraph. After the standard header, the format specifies *NumNets* the total number of nets and *NumPins* the total number of pins in the netlist. Then for each net specification starts with *NetDegree* the number of pins on this net and a net name. Then each line describes a pin corresponding to that net specifying *node name*, *pin direction*, *x-offset*, and *y-offset*. The node names are as in .nodes file. Direction of a pin can be either I(input), O(output), or B(bidirectional). For wirelength driven placement, the pin direction can be ignored. The x ad y-offset specifies the location of the pin on node(module), and calculated from center of that node. A sample .nets file is presented below.

```
# UCLA nets 1.0
# File header with version information, etc.,
# Anything following # is a comment, and should be ignored

# for each net
# NetDegree     : degree_of_this_net    net_name
#       node_name       pin_dir :       pin_xoffset     pin_yoffset

NumNets : 2             # Total number of nets
NumPins : 5             # Total number of pins in the netlist

NetDegree       :       3       n0
        o0      I       :        0.0000 -1.5000
        o1      I       :       -5.0000  0.5000
        p0      I       :        0.0000  0.0000
NetDegree       :       2       n1
        o3      O       :       10.5000 -1.5000
        o2      I       :       -1.0000  0.5000

# Pin offsets for each node are from the center of the node.
```

WTS file

It contains the weights on the nets. This file is not required for the our placement. So we assume the net-weights on all the nets to be equal (i.e. 1).

SCL file

This file represents configurations of individual rows in a Standard Cell Layout. Every row consists of non-overlapping subrows that are aligned at the coordinate of the row. Subrows can only differ by their origin and the number of sites. A site signifies the unit are for placement, thus a subrow is an array of such sites. A module(node) can obviously be more wide than a site, thus requires more than one site for its placement (height of a moveable module is either of same or may be of smaller height as of any subrow). There should be a feed-through gap between modules(nodes), width of this gap is given by site-spacing. A sample .scl file is presented below. The same information is depicted in figure 4.1.

```
# UCLA scl 1.0
# File header with version information, etc.,
# Anything following # is a comment, and should be ignored

NumRows : 2             # number of circuit rows for placement

CoreRow Horizontal
    Coordinate   : 32  # y-coordinate of the bottom edge of the row
    Height       : 16  # row-height
    Sitewidth    : 1   # width of each placement site
    Sitespacing  : 1   # spacing of each placement site
    Siteorient   : N
    Sitesymmetry : Y
    SubrowOrigin : 20   NumSites : 160
End
CoreRow Horizontal
    Coordinate   : 52  # y-coordinate of the bottom edge of the row
    Height       : 16  # row-height
    Sitewidth    : 1   # width of each placement site
    Sitespacing  : 1   # spacing of each placement site
    Siteorient   : N
    Sitesymmetry : Y
    SubrowOrigin : 60   NumSites : 120
End
# SubrowOrigin is x-coordinate of the left edge of the subrow
# NumSites is the number of placement sites in the subrow
# x-coordinate of right-edge of subrow\\
                           = SubrowOrigin + Numsites*Sitespacing
```

.

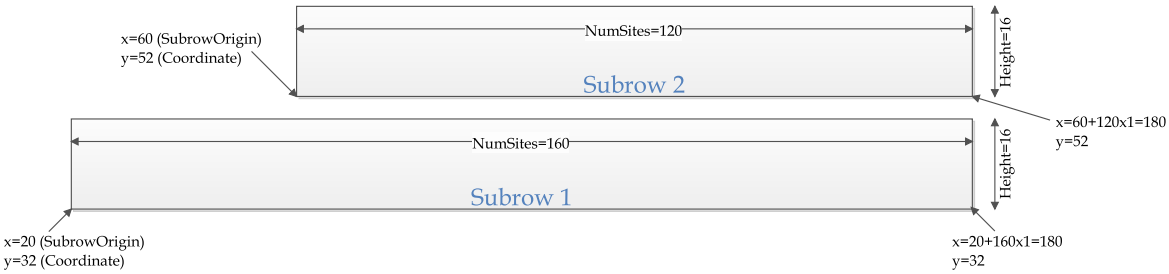

Figure 4.1: Visualization of the Standard Cell Layout

Siteorient and *Sitesymmetry* are optional, thus can be ignored for placement. Their details are given below. Orientations : the default orientation is "vertically and face up" - N (North). Rotate by 90deg clockwise to get E, S and W, flip to get FN, FE, FS and FW. (think of a dial). Symmetries: X and Y - allows flips around axis, ROT90 - allows any of rotations by 0, 90, 180 or 270 degrees

PL file

This file contains the (x,y)-placement of cells, and its orientation. The coordinates for all moveable nodes in the design will be $(0, 0)$ or undefined initially. The placer should parse this file to obtain the coordinates for the terminal/fixed nodes. A sample .nets file is presented below.

```
# UCLA pl 1.0
# File header with version information, etc.,
# Anything following # is a comment, and should be ignored

# node_name  ll_xcoord  ll_ycoord : orientation  movetype
        o0      0       0       :       N
        o1      0       0       :       N
        o2      0       0       :       N
        o3      7831    7452    :       N       /FIXED
        p0      1215    7047    :       N       /FIXED_NI
# ll_xcoord refers to lower left x coordinate and\\
                ll_ycoord refers to lower left y coordinate
# FIXED_NI refers to Not an Image which allows\\
                overlapping for such blocks
```

The PL file is also used for intermediate results and the final solution.

4.2 Storage structure

Our placer works fully based on main memory. Thus for large problem instance larger main memory is required. The parser module of our program reads the input files mentioned above and uses special data structures for storage.

1. The first one is a list that keeps all the nodes(modules) with its attributes namely name, width, height, current coordinate, and movetype. These data are obtained from .nodes and .pl files.

2. Whereas, the second one keeps the hypergraph information, i.e. it keeps a set of hyperedges, where each hyperedge is a set of nodes. To keep the relevant informations here we use a different type of node structure containing a reference to corresponding node, pin type, and pin offsets. These data are obtained from the .nets file.

3. Also, another list is required to keep the information of each row of the standard cell layout. These data are obtained from .scl file.

The usage of these data-structures are mentioned as required. The solution in each of the phases, i.e. updated coordinates of the modules, are reflected in the corresponding nodes for future use. Also a new .pl and .aux file is produced to make each module independent and pluggable.

4.3 Implementation of the Quadratic program

As shown is section 2.2 if we do not consider any non-overlapping constraint the quadratic program for the placement problem reduces to solving a system of linear equations. A system of n linear equations can be written as $AX = B$. Here A is an $n \times n$ *coefficient matrix* and the *right-hand side* vector B are known. The objective is to determine the *solution vector* X. If the diagonal elements of A satisfy the following condition then the system of equations is called *diagonally dominant*.

$$|a_{i,i}| \geq \sum_{\substack{j=1 \\ i \neq j}}^{n} |a_{i,j}|, \ \forall i = 1, 2, \ldots, n$$

If the inequality symbol replaced from $\geq$ to $>$ and the condition still holds for all $i = 1, 2, \ldots, n$; then the system of equations is called *strictly diagonally dominant*.

There are four major class of techniques to solve a system of linear equations:

1. Elimination method, also called direct method: Gauss-Elimination method try to eliminate x_1 from equation 2 through n, then it eliminates x_2 from equation 3 through n and so on. Now we find x_n from n-th equation, then find x_{n-1} from $(n-1)$-th equation by substituting the value of x_n. Doing this successively, values of all the unknowns are obtained.

2. Iteration method, also called indirect method: In Gauss-Jacobi method we try to find successive better approximation of the values of the unknowns. It express successive value an unknown based on its previous value. In each iteration we try to minimize the error. The number of iteration required depends upon the desired degree of accuracy. The system must be rewritten into diagonally dominant form (if not already), to ensure convergence. The Gauss-Seidel method is same as the Gauss-Jacobi method except in the iteration formula. Here the system must be rewritten into strictly diagonally dominant form (if not already), to ensure convergence.

3. Matrix-Determinant based method: The Cramer's Rule and Matrix inversion method tries to directly find the solution. Though these methods are very common in practice, both of these methods are tedious as they require to evaluate a number of determinants. These methods works only when A^{-1} exists i.e. when $|A| \neq 0$ (A is non-singular).

4. Matrix Factorization method: The LU decomposition method tries to write the equation $AX = B$ in the form $LU = PB$ where $PA = LU$. We now solve $LY = PB$ for Y and then solve $UX = Y$ for X. Note that in both cases we are dealing with triangular matrices (L and U) which can be solved directly by forward and backward substitution without using the Gaussian elimination process (however we do need this process or equivalent to compute the LU decomposition itself). This procedure is really efficient for finding solution of the system of simultaneous equations and requires $O(n^3)$ floating

point operations. The above procedure can be repeatedly applied to solve the equation multiple times for different b. In this case it is faster (and more convenient) to do an LU decomposition of the matrix A once and then solve the triangular matrices for the different b, rather than using Gaussian elimination each time. Here again the procedure won't work if the original A matrix is singular.

Thus we have used LU decomposition based method to solve the equations $Qx + d_x = 0$ and $Qy + d_y = 0$ obtained in section 2.2. For this we have used *DecompositionSolver* interface form The Apache Commons Mathematics Library. For more details refer to section A.1 in appendix. First, we have enumerated the connectivity matrix from the hypregraph informations, which is then used to obtain the Q matrix. The d_x and d_y vectors are obtained using the same connectivity matrix and the coordinates of the fixed modules available from the node list. Also we would like to state that, the Q matrix will always be diagonally dominant, thus one can use any of the solver method mentioned above.

4.4 Implementation of the Nonlinear program

In section 2.3.3 we obtained an unconstrained objective function to be minimized as given below.

Minimize $\sum_{e \in E} c_e \times LSEWL_{e,\alpha}(x_1, ..., x_n, y_1, ..., y_n) + \beta \times \sum_{\substack{i,j \in V \\ i \neq j}} Overlap_{ij}(x_i, y_i, x_j, y_j)$

As it is evident that, the objective function has two sub-functions and both are defined in terms of log-sum-exponentiations. Such a nonlinear multivariate (involving many variables) expression can be minimized using **conjugate gradient method**. For this we have used *NonLinearConjugateGradientOptimizer* interface form The Apache Commons Mathematics Library. For more details refer to section A.2 in appendix. We have used $\alpha = 3000$ and tuned for various values of β. The solution obtained by the quadratic programming is used as the initial guess for the Conjugate Gradient method.

4.5 Implementation of the Legalization

In chapter 3 we described the legalization process and the Tetris algorithm. The implementation of the algorithm straight forward, we simply need to convert each line of the algorithm to equivalent Java statements. The solution obtained by the nonlinear optimizer is used for legalization by the Tetris algorithm.

4.6 Implementation of the GUI

A graphical user interface (GUI) has also been implemented to visualize flow of the placer algorithm in various phases. The *Java Swing API* has been used to develop

the GUI, which required only a few lines of code. Our GUI is a simple one, thus cannot handle large real life problem instance (the modules will go beyond the view-port), although the placer algorithm in background should correctly.

4.7 Executing the placer

The Java Runtime Environment is required for execution of our program. Our placer program can be used as follows:

java −jar placer.jar

It will prompt two times for the input and output file names. For the input file enter the name of AUX file, and for the output file enter a name for PL file. Filenames can come along with its path (if required). A sample run on the above data is shown below.

```
C:\Users\RATHIN\Desktop>java -jar placer.jar

Enter aux file name: sample data\data0\circuit.aux
Starting the parser...
Parsing is done.
Enter output file name: sample data\data0\a.pl
Starting the Quadratic solver...
Finished.
Starting the Nonlinear solver...
Finished.
Leagalizing the solution...
Finished.
The final HPWL: 81.0
Total overlap: 0.0
Output file wirtten.
```

The contents of the output file is shown below.

```
# UCLA pl 1.0
# node_name  ll_xcoord  ll_ycoord : orientation  movetype
        o0      50.0    32.0    :       N
        o1      45.0    32.0    :       N
        o2      20.0    32.0    :       N
        o3      31.0    52.0    :       N       /FIXED
        p0      15.0    47.0    :       N       /FIXED_NI
```

To invoke the GUI type the following:

java –jar placer.jar –gui

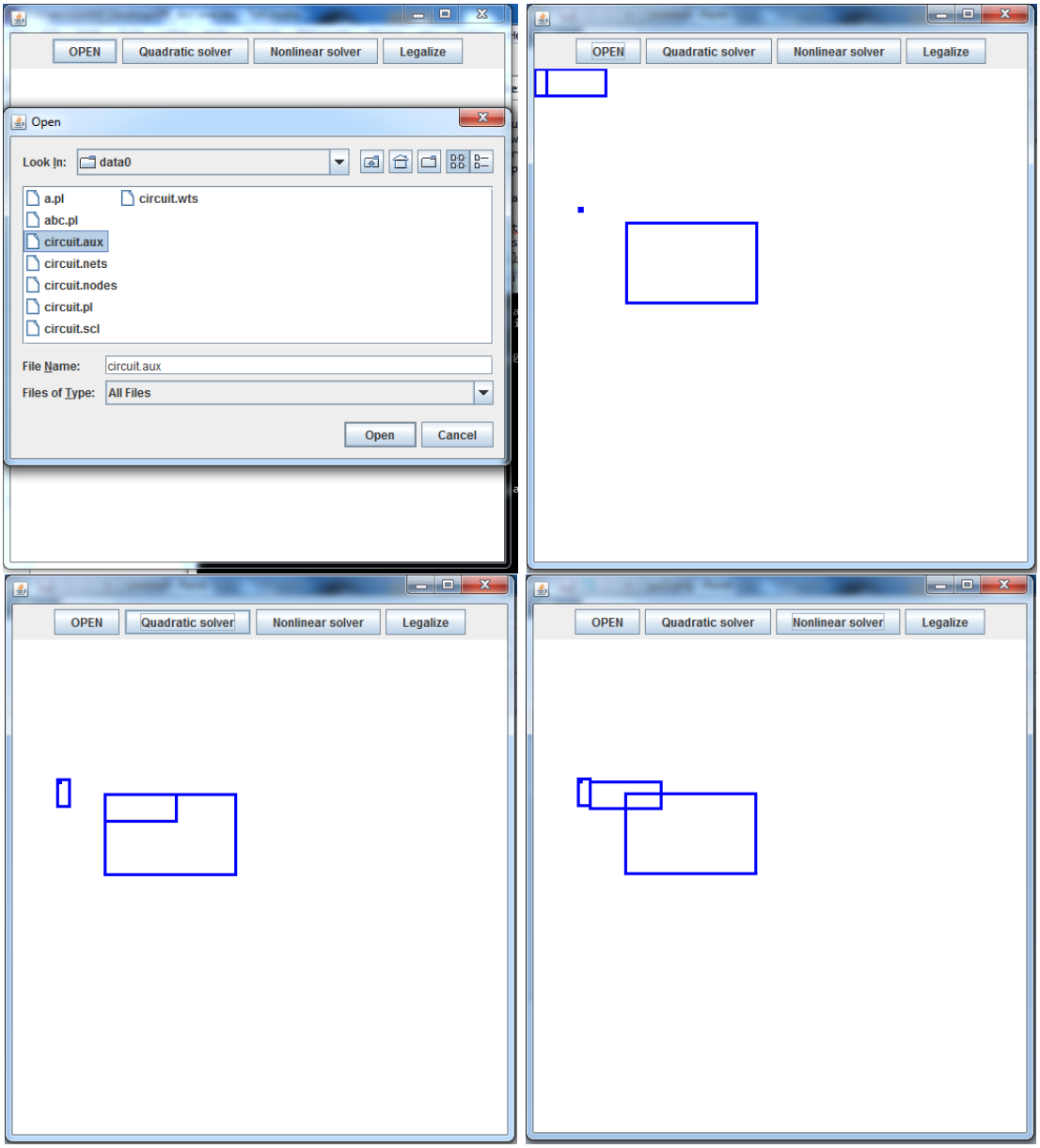

Figure 4.2: Execution using GUI

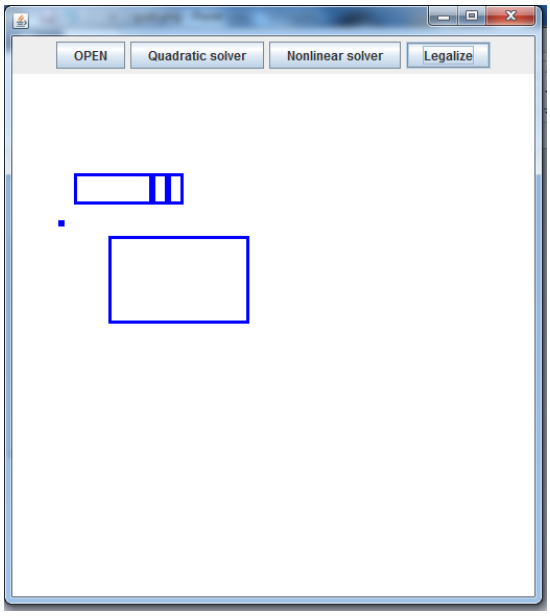

Figure 4.2: Execution using GUI (contd.)

Future Scope

We have implemented a simpler version of the placer program. In order to handle real life large data numerous modifications needs to be done. As we have stated earlier, that our placer is fully primary memory based, it can not handle arbitrarily large data set. For example, the Quadratic program requires a large square matrix for its implementation which is in the order of $O(r^2)$, where r is the number of movable modules. To tackle this kind of challenges, most placer algorithms first apply a *partition based technique* to make the data set manageable. This kind of algorithms can be explored in the future. Also our nonlinear program can be improved significantly by using *density based non-overlapping constraints*. Moreover, *detailed placement* techniques can also be implemented on top our placer program to further improve the solution.

Appendix A

The Apache Commons Mathematics Library

Commons Math is a library of lightweight, self-contained mathematics and statistics components addressing the most common problems not available in the Java programming language. This library has certain benefits like:

- This package emphasizes small, easily integrated components rather than large libraries with complex dependencies and configurations.
- All algorithms are fully documented and follow generally accepted best practices.
- No external dependencies beyond Commons components and the core Java platform (JDK 1.5+).
- It is freely distributed under the terms of the Apache License, Version 2.0 and available at: `http://commons.apache.org/proper/commons-math/`

Currently (as of 15th August, 2016) we are using Version: 3.6.1 of Commons Math. In the following sections we describe how to use the this library to solve a system of linear equations and optimize a nonlinear objective function.

A.1 Solving a system of linear equations

It is an interface for handling decomposition algorithms that can solve a system of linear equations of the form $AX = B$. For this we need two additional types:
1. `RealMatrix`: it an interface supplied by the Commons Math to handle a matrix with real numbers as entries and support elementary matrix operations. It can be instantiated in multiple ways, e.g, we can use the `BlockRealMatrix` subtype as it is cache friendly implementation of `RealMatrix`:

```
//matrixData of type double[][] contains the values of cells
RealMatrix mat = new BlockRealMatrix(n, m);
for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
                mat.setEntry(i, j, matrixData[i][j]);
        }
}
```

2. `RealVector`: this interface represents a vector with real numbers as entries and support elementary vector operations. It can be instantiated in multiple ways, e.g, we can use the `ArrayRealVector` subtype:

```
//vectorData of type double[][] contains the values
RealVector v = new ArrayRealVector(vectorData, false);
```

The second parameter, `copyArray`, is set to false. This will prevent the copying and improve performance as no new array will be built and no data will be copied.

Now we are in position to use the solver. Solving the system is a two phases process: first the coefficient matrix is decomposed in some way and then a solver built from the decomposition that solves the system. This allows to compute the decomposition and build the solver only once if several systems have to be solved with the same coefficient matrix. For example, to solve the linear system

```
2x + 3y - 2z = 1
-x + 7y + 6x = -2
4x - 3y - 5z = 1
```

We start by decomposing the coefficient matrix A (in this case using LU decomposition) and build a solver

```
double[][] coefficients = new double[][] { { 2, 3, -2 },
                                           { -1, 7, 6 },
                                           { 4, -3, -5 } };

RealMatrix A = new BlockRealMatrix(3, 3, coefficients, false);

DecompositionSolver solver = new LUDecomposition(A).getSolver();
```

Next we create a `RealVector` array to represent the constant vector B and use solver to solve the system.

```
RealVector constants = new ArrayRealVector(
                                new double[] { 1, -2, 1 }, false);

RealVector solution = solver.solve(constants);

double x,y,z;
x = solution.getEntry(0);
y = solution.getEntry(1);
z = solution.getEntry(2);
```

The LU Decomposition solver requires coefficients matrix to be a square matrix and gives exact solution. There are other solver methods namely Cholesky, QR, eigen decomposition, and SVD.

A.2 Optimizing a nonlinear function

As it has already been stated that, a nonlinear multivariate (involving many variables) expression can be minimized using **conjugate gradient method**. An algorithm for the same is also presented in section 2.3.3. The algorithm needs to compute gradient, i.e., we need to obtain partial derivatives of our function. For this we are again using Commons Math library. To obtain the solution the following needs to be done.

Firstly, we need to write our objective function (which we want to optimize) by implementing the `MultivariateDifferentiableFunction` interface. This will require to implement the two unimplemented methods. The two method basically encodes the same thing (the value of our objective function at a given point), but in different way. The first one is simple and straightforward to write. Whereas, the second one uses `DerivativeStructure`.

The class `DerivativeStructure` is used to represent both the value and the differentials of a function which automatically computes the values of higher order derivatives. The `DerivativeStructure` is basically an array like structure. If we specify to store derivatives of an n variable function upto order 2, then the first location will contain the value, the next n locations will contain the values of 1st order partial derivatives with respect to each of the variables, and next n location will have the values of 2nd order partial derivatives with respect to each of the variables. Suppose we are trying to minimize the function:

$$f(x_1,\ x_2) = e^{(x_1-2)^2} + e^{(x_2-3)^2}$$

. It can be easily verified that the minima will be attained when $x_1 = 2$ and $x_2 = 3$. We now define this function f as a class called `myFunction` as follows.

```
public class myFunction
                implements MultivariateDifferentiableFunction{

        @Override
                public double value(double[] point) {
                double a = point[0] - 2;
                double b = point[1] - 3;
                double x = Math.exp(a*a)+ Math.exp(b*b);
                return x;
        }

        @Override
        public DerivativeStructure value(DerivativeStructure[] point)
                        throws MathIllegalArgumentException {
                DerivativeStructure a = point[0].subtract(2);
                DerivativeStructure b = point[1].subtract(3);
                DerivativeStructure a2 = a.multiply(a);
                DerivativeStructure b2 = b.multiply(b);
```

```
                DerivativeStructure a2e = a2.exp();
                DerivativeStructure b2e = b2.exp();
                DerivativeStructure r = a2e.add(b2e);
                return r;
        }
}
```

Now need to optimize the function, for this we will use a nonlinear optimizer class called `NonLinearConjugateGradientOptimizer`. Currently there exists two implementation of the class, we are using the one from the package `org.apache.commons.math3.optimization.general`. This class supports both the Fletcher-Reeves and the Polak-Ribire update formulas for the conjugate search directions. It also supports optional preconditioning.

First we need to create an instance of `NonLinearConjugateGradientOptimizer`. The constructor of this class needs two parameters, first one of the update formula for the conjugate search directions, and the second one is for convergence checker where we are passing the error threshold.

```
NonLinearConjugateGradientOptimizer optimizer = new
                NonLinearConjugateGradientOptimizer(
                        ConjugateGradientFormula.POLAK_RIBIERE,
                        new SimpleValueChecker(1e-6, 1e-6));
```

Now we need to optimize our function. To do this we call the `optimize()` method of the object `optimizer`, and pass the following parameters. The first one is the maximum number of iterations to be evaluated, second one is an instance of our objective function, third one is the type of optimization (either minimize or maximize), and the last one is our initial guess for the optimum values.

```
double[] initialGuess = new double[] {1.8, 3.2};
PointValuePair val = optimizer.optimize(100, new myFunction(),
                        GoalType.MINIMIZE, initialGuess);
```

Bibliography

[1] N. A. Sherwani, *Algorithms for VLSI Physical Design Automation.* Kluwer Academic Publishers, 3rd ed., 2013.

[2] "Physical design, https://en.wikipedia.org/wiki/physical_ design_ (electronics)."

[3] L. Wang, Y. Chang, and K. Cheng, *Electronic Design Automation: Synthesis, Verification, and Test.* Systems on Silicon, Elsevier Science, 2009.

[4] W. Naylor, R. Donelly, and L. Sha, "Non-linear optimization system and method for wire length and delay optimization for an automatic electric circuit placer," Oct. 9 2001. US Patent 6,301,693.

[5] A. B. Kahng, S. Reda, and Q. Wang, "Aplace: A high quality, large-scale analytical placer," 2007.

[6] D. Hill, "Method and system for high speed detailed placement of cells within an integrated circuit design," Apr. 9 2002. US Patent 6,370,673.

[7] "Generic hypergraph formats, http://vlsicad.ucsd.edu/gsrc/bookshelf/slots/fundamental/hgraph/hgraph1.1.html."

[8] "Placement formats, http://vlsicad.ucsd.edu/gsrc/bookshelf/slots/placement/plformats.html."