
# VIRTUAL REAL-TIME SIMULATION OF A COMPLEX PRODUCTION LINE

Optimization of Complex Automated
Production Lines
Applied On a Sheet-Metal Press Line

Yogesh H. Jain
(yogeshj@me.iitb.ac.in)

Supervisors: Dr. Fredrik Danielsson
Mr. Bo Svensson

**May – July 2007**

# Abstract

This paper describes my work done at University West, Trollhättan, Sweden as the part of Summer Internship during the period May – July 2007.

The prospective work is to optimize the Production Lines for Sheet Metal Forming. The optimization part is then implemented in the Off-line Control Software, PRESSOPT. In this project an optimization method for highly automated industrial production lines for the Manufacturing Unit of Volvo, Gothenburg, Sweden is investigated. The main aspect of the work is to optimize the complex control system applications where it is not possible to create a useful simplified model representation of the control system or for performing the complete state space search.

The aim of this work is to achieve a higher production rate without any collisions of the moving part. The value of the parameters on which the function depends on is evaluated in the feasible region using Nelder Mead Method. Nelder Mead, first published in 1965, is a very popular Direct Search method for multidimensional unconstrained minimization problems without using derivatives; and no theoretical proofs are present till date. For this Nelder Mead process of optimization needs to be implemented and tested on the off-line control software, PressOpt. This project helps in implementing Nelder Mead in PressOpt.

## TABLE OF CONTENTS


**1. INTRODUCTION ..... 4**

I. PROJECT DETAILS: ..... 4
II. VRTM: ..... 5

**2. WORK DESCRIPTION ..... 7**

I. SCREENING METHOD: ..... 8
II. NELDER MEAD METHOD: ..... 9

**3. IMPLEMENTATION OF NELDER MEAD IN PRESSOPT ..... 11**

I. FUNCTION.CPP ..... 11
II. NELDERMEAD.CPP ..... 15

**4. INFERENCES ..... 22**

I. RESULTS: ..... 22
II. ADVANTAGES OF NELDER MEAD ..... 22
III. SOME PROBLEMS WITH NELDER MEAD: ..... 23
IV. FUTURE ASPECTS AND OTHER METHODS: ..... 23

**5. ACKNOWLEDGEMENT ..... 24**

**6. REFERENCES ..... 25**

# 1. Introduction

## I. Project Details:

Maximizing the lifetime throughput, i.e. increasing the production rate with optimum usage of available resources is the all-time requirement of each and every manufacturing plant. In order to attain a high lifetime throughput, it is important not only to have a high throughput rate during production but also to minimize production stops, both in number and time. Therefore, an optimization strategy often required to reach the production goals. In highly automated production lines the optimization process mainly involves control code strategies, software development and parameter tuning. Today, optimization in automated production line is normally performed on-line and is based on the line operators and control engineers experience and capacity. Unfortunately, this empirical based method is predominant and widely used because of lack of useful industrial methods. Even if there exists method in other industrial fields, e.g., factorial design, it might be difficult or even impossible to apply these methods on automated production lines since most of these methods requires extensive and in some case extreme experiments to be carried out and evaluated. Even if an optimized production rate can found it is not motivate due to the cost and effect of such experiments.

In this project an optimisation method for highly automated industrial production lines is investigated. The key issue here is to optimize complex control system applications where it is not possible to create a useful simplified model representation of the control system or to perform a complete state space search. A complex control system installation today might exist of more than 100000 lines of code, complex interlocking together with thousand of distributed I/Os and extensive communication with other parts. This type of complex applications lacks for feasible optimisation methods. The result of this project is a working optimisation method where complex parameterised control system functions can be optimized.

This project deals with an optimization method called Nelder Mead Method of Direct Search for multidimensional and unconstrained problems. Nelder Mead Method does not uses derivatives to find the global minimum value of the objective function and is hence easy to implement in the code and in practical sense. Nelder Mead method finds the global minimum value of the function without any constraints to it. Compared to the other optimization methods implemented, like Full Factorial Method, Nelder Mead method searches for optimum

values of parameter at each and every feasible region; there is no need to specify any boundaries to it. While in full factorial method the best parameter value is obtained only within the boundaries specified by the parameter's low and high values.

In this paper the complete details of my project work during the Internship period is being explained, which includes, how the Nelder Mead method is implemented and used and its drawbacks if any, along with the comparison with the full factorial method that I had came across while on work.

## II. VRTM:

For the above purpose of maximizing production rate and efficiency of the manufacturing line there is a need for tools and methods to develop the control software off-line to manage programming, verification, validation, optimization and training new operators without using the real production line for these. Hence a Virtual Real Time Model (VRTM) of the existing line for simulation of industrial production lines is required.

VRTM is intended to be used for off-line control software development and is aimed to increase the production rate at its optimum level. The main advantage in using VRTM is that the same feature is being used in the real time production line system and also that the virtual time is used and everything is time-synchronized and works similar to the existing real time systems including the security functions.

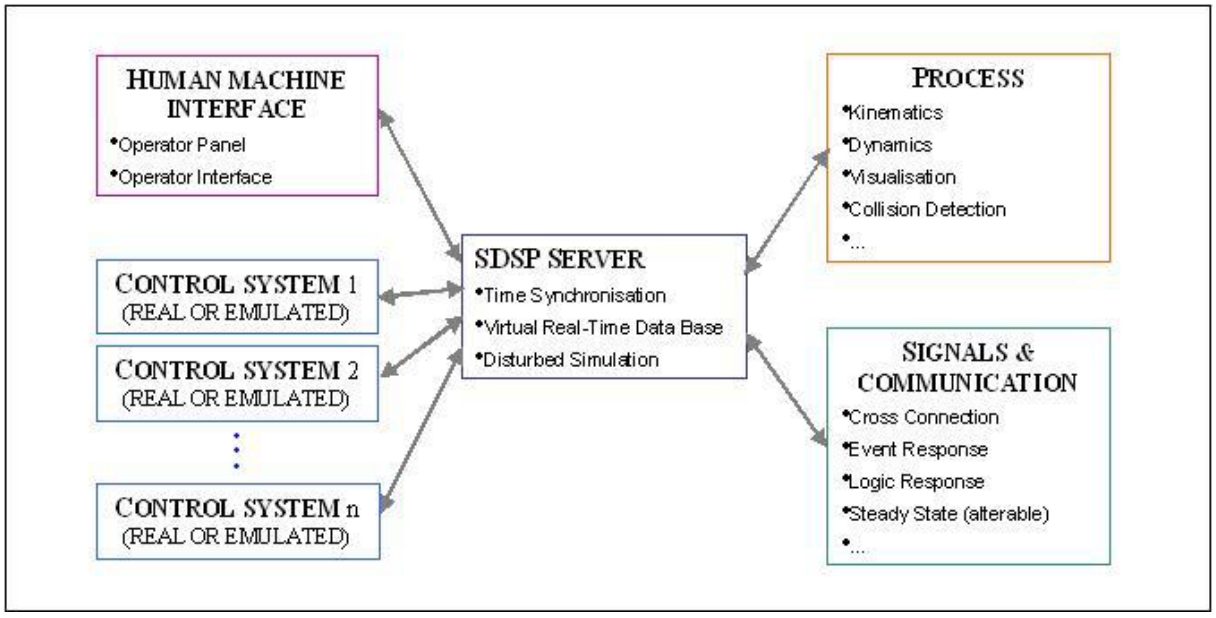


**Figure 1:General Virtual Real-Time Model**

For the off-line optimization using VRTM the model includes the study of Parameters and there effects, simplified performance model and then the optimization method like Nelder Mead. The various parameters on which the life-time throughput rate depends on are:

P0: Release Loader Start
P1: Release Loader Stop
P2: Release Press Start
P3: Release Press Stop
P4: Release Unloader Start (Press)
P5: Release Unloader Stop (Press)
P6: Position Wait
P7: Position Low Speed
P8: Position Home
P9: Speed Fetch-Wait
P10: Speed Wait-Leave
P11: Speed Leave-LowSpeed
P12: Speed LowSpeed-Home
P13: Speed Home-Fetch

Of all these 14 parameters P1, P3, P5 and P6 are kept constant and others are operator adjustable parameters.

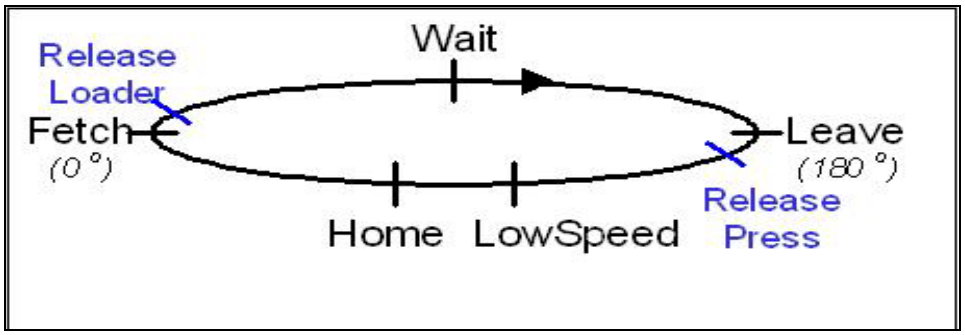


**Figure 2: Unifeeder n**

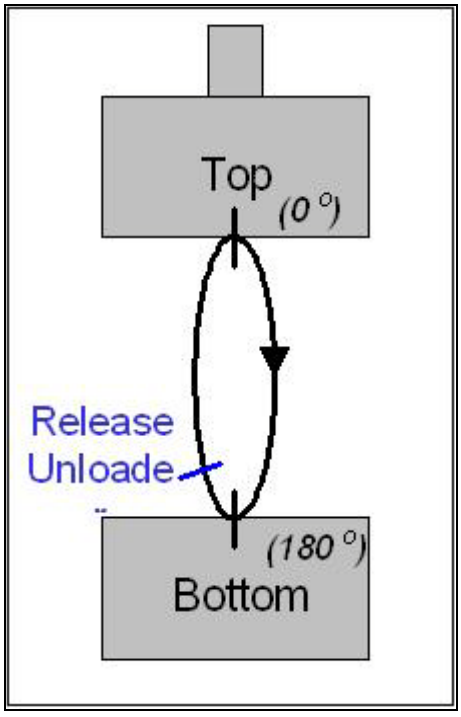


**Figure 3: Press n**

# 2. Work Description

This Project was defined to optimize the operator controllable parameter values out of 14 parameters so that we obtain maximum production rate for Sheet Metal Forming in Press Lines and no collisions between any moving parts are seen. If a collision occurs, it is detected by Rob Cad and is reported to VRTM. To avoid any such collisions, an optimizer controls the Mean Accelerations with plate and Mean Acceleration without carrying plate by minimizing the function, as fast moving robotic hands (unifeeder) increases the risk of dropping of plates. Hence, the function that is required to be optimized is:

***f*(P)** = k1****f*(Prod Rate)** + k2****f*(Mean Acc with plate)** + k3****f*(Mean Acc without plate)**

Where k1, k2 and k3 are weighted constants. This function needs to be optimized since we require maximum Production Rate and minimum Mean Accelerations. Hence, generally, the constants are assigned values of -1, 0.6 and 0.2 respectively.

The work assigned was to build an Optimizer that works like:

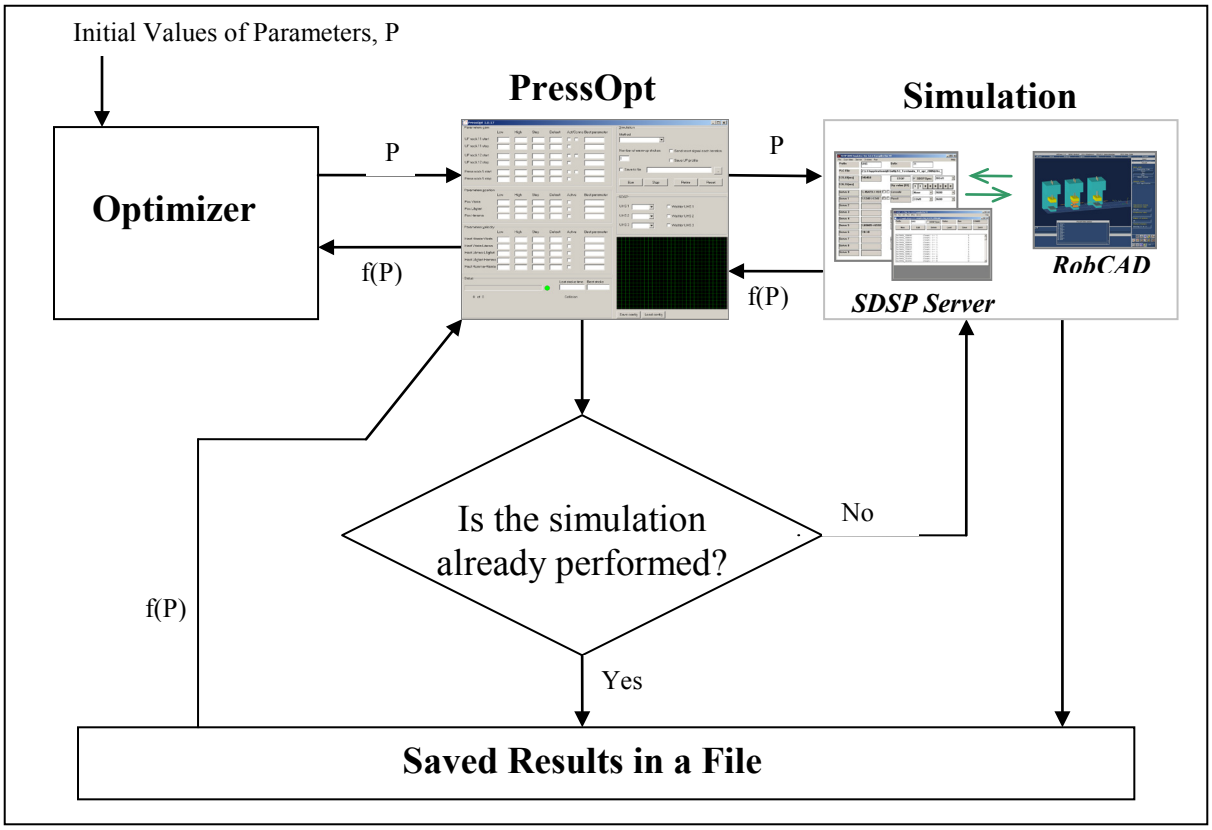


**Figure 4: Process Diagram.**

Initially and currently "Screening Method" is used in Optimizer to look at the optimum value that can be obtained in the range of Low and High values specified but only at the specific steps mentioned. Hence we weren't sure whether there exists a point beyond the specified

range where an optimum value lies. To search such point we were in need for a Global Search method which can do the job with any intricacy, and hence one of the solutions was to use Direct Search Method. Nelder Mead method is one such Direct Search Method which does not need any constraints but finds only the minimum value of the function.

# I. SCREENING METHOD:

**Screening Method** uses **Two Full Factorial Method** of point selections. In Screening Method a point is selected by different combinations of steps in the pre-specified range of the parameters. The range of a parameter is defined by its Low and High value. For within this range Step is also defined for selection of points in the increment of step from low to high with all combinations possible with other parameters.

For example, working with 2 active parameters (active here stands for those parameters which we want them to control the results and is selected by user interface Press Opt Window) we have 4 combinations that need to be simulated, if step is equal to the difference of high and low.

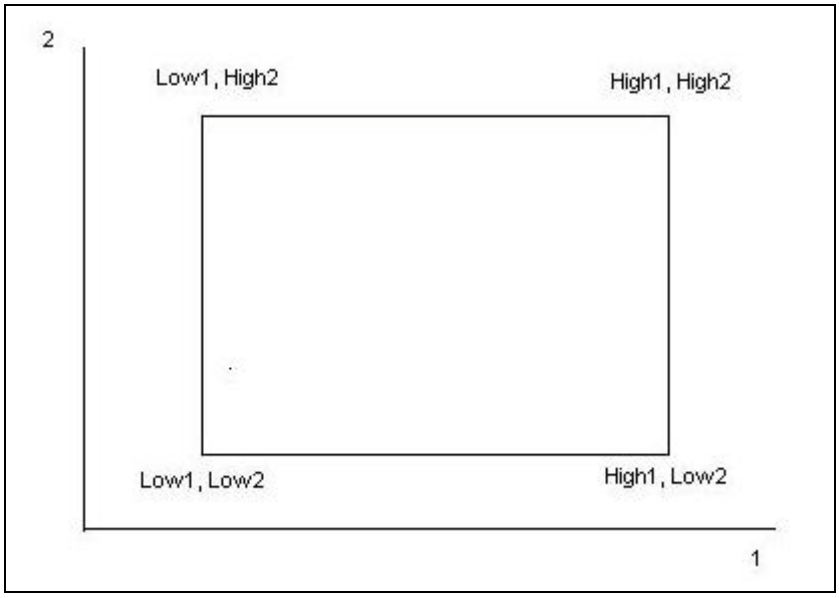


**Figure 5: Screening method for 2 variables**

Hence while optimizing the problem arises if the optimum point is beyond the limit mentioned. Though we can increase the limit we can not reach to the next point obtained by combinations other than linear way. Hence if the point, (say (High1, Low2) finds a collision then the method would suggest that we cannot reach to other point other than (High1, High2) and greater than High1 value where an optimum might exist. Also controlling 10 parameters

in single simulation process requires minimum of 1024 strokes to reach optimum value, which obviously is time consuming.

To overcome some of the problems faced while using Screening Method, Nelder Mead method was suggested, as it is simple to implement a method which does not involve derivatives.

## II. NELDER MEAD METHOD:

**Nelder Mead Method** is a Simplex based Direct Search Method. **Direct Search Methods** uses comparisons of the values of the objective function and do not involve its derivatives. A **Simplex** in $R^n$ is a set of n+ 1 point that does not lie in hyper-plane (In 2D the simplex is a triangle). Nelder Mead Method involves 4 basic steps:

i. Reflect
ii. Expand
iii. Contract
iv. Shrink

Nelder Mead method works in finding minimum value of the function. The function value is evaluated at every initial points/vertices and is then arranged in the increasing order of the function value. The worst vertex, at which the function has largest value, is rejected and is replaced by the newly found vertex and the process continues under the conditions required and using basic steps mentioned above.

In the process following terminologies are used:

- **B**: Best Point, or point with minimum function value.
- **W**: Worse point or point with maximum function value.
- **G**: good point or point with second largest function value.
- **M**: Mean point of all points except point W.
- **R**: Reflection point of W about M
- **E**: Point obtained by expanding point R in direction of W-M-R.
- **C**: Point obtained by Contracting W in the direction of M.

The process proceeds as shown below:

- Evaluate and Arrange the Vertices $f(x1) < f(x2) < \ldots\ldots < f(xn+1)$
- Find Mean vertex, **M** of all points except the worst vertex W$(xn+1)$
- Reflect W about M to have **R**. evaluate $f(R)$
- If $f(R) < f(G)$ (point G has the second largest value)

If ***f*(B) < *f*(R)** replace W with R and start the process all over gain.

Else, **Expand** or move in the right direction so obtained to find point E. and at last replace W with E and arrange to find other best point than E.

- Else **Contract** to find point C.

  If ***f*(C) < *f*(G)** replace W with C and continue the process from beginning.

  Else

  If ***f*(R) < *f*(W)** then update W with R

  Else {do nothing}

  If ***f*(C) < *f*(W)** update W with C.

  if ***f*(C) = *f*(W)** then **Shrink** the Simplex such that the distance between a vertex and Best point is reduced to half and Start the process all over again.

- The iteration process continues till the difference between ***f*(W)** and ***f*(B)** is under the specified user range or if user defined number of iterations are done. For example: if **|*f*(W)-*f*(B)| < 0.1 or NumberOfIteration = 15** then "STOP".
- Find the minimum value of all the so obtained new points and that would be the Optimum value.

However no such proofs of its convergence exists till date. Hence implementation of this method in one way or the other can prove this method practically and physically. And will also help us to know any of the problems or drawbacks that this method faces.

# 3. Implementation of Nelder Mead in PressOpt

Implementation of Nelder Mead method is braked up into two parts: first part consists of all the basic methods the code requires while performing the iteration, while the second part consists of defining "neldermead" structure which is called in 'simulationthread.cpp'.

## I. FUNCTION.CPP

Performing Nelder Mead for N+1 point in $R^n$ space requires an easy way to fill in and store values of initial points which later on is updated as required. Hence, use of Matrix of size (14, N+1) simplifies our job. Note that we also require the Optimizer to know whether the set of Initial points was being called previously or not, and hence checking a Matrix makes our job much easier. Hence a structure Matrix is defined as follows:

```
struct Matrix
{
        double** data;
        int rows;
        int cols;
};

Matrix matrix_malloc(int rows, int cols)    // Allocation of matrix
{
        int i;
        Matrix A;
        A.rows = rows;
        A.cols = cols;
        A.data = new double*[rows];
        for(i =0; i<rows; i++)
          A.data[i] = new double[cols];

        return A;
}
```

Since in a single stroke for simulation a set of all 14 parameters are required we need to have a column vector that defines the Vertex or the point at which the simulation would be operating.

Matrix related methods are also defined for operating on matrix. Examples:

| Function | Methods | Returns |
|---|---|---|
| Extract column vector from Matrix | Matrix get_vector(Matrix A, int col) | Column Vector of size (A.rows,1) |
| Set each elements of matrix equal to "value" | Matrix set_matrix(Matrix A, int value) | Matrix B with B.data[i][j] = value |
| Print Matrix | void get_matrix(Matrix A) | |
| Addition of two matrices: | Matrix addition(Matrix A, Matrix B) | Matrix C = Matrix A + Matrix B |
| Subtraction of two matrices | Matrix subtraction(Matrix A, Matrix B) | Matrix C = Matrix A - Matrix B |
| Division of matrix with scalar | Matrix divide(Matrix A, double value) | Matrix C = (Matrix A)/value |
| Multiplication with scalar | Matrix multiply(Matrix A, double value) | Matrix C = (Matrix A)*value |
| Transpose of Matrix | Matrix T(Matrix A) | Matrix C (A.cols, A.rows) C.data[j][i] = A.data[i][j] |

With these basic matrix methods other iteration related methods are also defined in this file. They are as follows:

| Function | Methods | Returns |
|---|---|---|
| To find Min value in the row vector (Best Point, B) | Matrix best(Matrix Z) | Matrix B with B.data[0][0] = min value B.data[0][1] = column (position of the value) |
| To find Max / largest value in the row vector (Worse Point, W) | Matrix worst(Matrix Z) | Matrix W with W.data[0][0] = max value W.data[0][1] = column (position of the value) |
| To find second largest value in the row vector (Good Point, G) | Matrix bad(Matrix Z) | Matrix G with G.data[0][0] = value G.data[0][1] = column (position of the value) |

| | | |
|---|---|---|
| Finding Mean point | Matrix Mean(Matrix A, int col)<br>*Note: **col** indicates the position of Worse Point in matrix A* | Column vector M of size (A.rows, 1) |
| Reflection | Matrix Reflection(Matrix M, Matrix Aw, double alpha)<br>*Note: **Matrix Aw** is the worse point And **alpha** is chosen by User.* | Reflection of Aw about M. Column vector R of size (A.rows, 1) |

The calculation of function value and updating of collision value are also done within this file by using the method

```
Matrix f(Matrix A, Matrix* Collide)
{
	Matrix B=matrix_malloc(2,A.cols);
	int collision;
	for(int i=0;i<A.cols;i++)
	 {
		collision = (int)(*Collide).data[0][i];
		get_vector(A, i);
		B.data[0][i]= SimulateOneStroke(&collision);
		B.data[1][i]= collision;
	 }
	return B;
}
```

This method returns a Matrix of size (2, A.rows) where the first row gives the function values of each column while the second row provides the collision value. Note SimulateOneStroke(*int CollisionFlag) is already defined method in 'simulation.cpp' and it returns the answer after performing the simulation. The 'answer' is of the form:

**answer = k1*glRun.ProdRate + k2*glRun.MeanAccComp +k3*glRun.MeanAccEmpty**;

where glRun.ProdRate gives Production Rate, glRun.MeanAccComp gives mean acceleration of robotic hand with plate and glRun.MeanAccEmpty stands for mean acceleration of the robotic hand not carrying plates and the constants k1, k2 and k3 are assigned values of -1, 0.6 and 0.2 respectively.

The methods are also defined for the basic steps of Expansion, Contraction and Shrinkage.
In Expansion, Point E is found by expanding R to 'gamma' times its distance from M in the same direction and the process continues to find new point E such that with old point E as new point R if the function value at new point E is less than the function value at old E.

```
//--------------------Expansion-------------------------//
Matrix Expansion(Matrix R, Matrix M, double gamma)
{
    Matrix E = matrix_malloc(R.rows, 1);
    Matix Ze=matrix_malloc(2,1);
    Matrix Zr=matrix_malloc(2,1);
    Matrix CollideR = matrix_malloc(1,1);
    Matrix CollideE = matrix_malloc(1,1);
    Zr = f(R, &CollideR);
    E = addition(multiply(R,gamma),multiply(M,(1-gamma)));
    Ze = f(E, &CollideE);
    double Zeold = Zr.data[0][0];
    int j = 1;
    while(Ze.data[0][0] < Zeold)
     {
          Zeold= Ze.data[0][0];
          E = subtraction(multiply(R,gamma+j),multiply(M,(j+gamma-1)));
          Ze = f(E, &CollideE);
          j++;
     }
    return E;
}
```

In Contraction, point C is found by reducing the distance between worse point either W or R and mean point M by the factor of 'beta'.

```
//--------------------Contraction-------------------------//
Matrix Contraction(Matrix Aw, Matrix R, Matrix M, double beta)
{
     Matrix C=matrix_malloc(R.rows,1);
     Matrix C1=matrix_malloc(R.rows,1);
     Matrix Zc=matrix_malloc(2,1);
     Matrix Zc1=matrix_malloc(2,1);
     Matrix CollideC = matrix_malloc(1,1);
     Matrix CollideC1 = matrix_malloc(1,1);
     C = addition(multiply(Aw,beta),multiply(M,(1-beta)));
     Zc = f(C, &CollideC);
     C1 = addition(multiply(R,beta),multiply(M,(1-beta)));
     Zc1 = f(C1, &CollideC1);

     if(Zc1.data[0][0] < Zc.data[0][0] && Zc1.data[1][0]== 0)
          for(int j=0; j<R.rows;j++)
             C.data[j][0] = C1.data[j][0];

     if(Zc.data[1][0]!=0)
          for(int j=0; j<R.rows;j++)
             C.data[j][0] = C1.data[j][0];
     return C;
}
```

While in Shrinkage, distance between point B and all other points is reduced to half and it updates the original matrix A.

```
//----------------------Shrinkage----------------------------//
void Shrink_update(Matrix *A, Matrix Ab)
{
       int i,j;
       for(i=0;i<(*A).rows;i++)
          for(j=0;j<(*A).cols;j++)
             (*A).data[i][j] = (Ab.data[i][0]+ (*A).data[i][j])/2;
}
```

And at last there are two methods that too are required during iterations.

```
//-------------------- Updating Original Matrix ------------//
void update(Matrix* A, Matrix R, int W)
{
        for(int j=0; j<(*A).rows;j++)
          (*A).data[j][W] = R.data[j][0];
}
```

This method updates original Matrix A by inserting column vector Matrix R at the column "W". And the final method **double Abs(double x)** is used to calculate the magnitude of the error 'x'.

## II. NELDERMEAD.CPP

This file contains two methods: the first one is defined to call this method from 'simulationthread.cpp' file and the second one is used for iteration process.

The first method initializes the nelder mead process by accepting the initial starting vertices. This could be done either by having another method that uses the pre-defined low and high values in user interface and gives different combinations and then out of these combinations best N+1vertices can be chosen. Also the start values can be entered by inserting a file through which the method understands the matrix elements of matrix A which contains these starting vertices. And once the initialization is done the iteration process is called, which returns 1 or -1 depending on the number of iterations or error comparison. This governs the while loop in the simulationthread.cpp file.

```
long neldermead(int Why, long *Answer, tParams Parameters[], int *CollisionFlag)
{
//----------------Determining Dimension to Work on.----------//
  for(i=0; i!=NUMBER_OF_PARAMS; i++)
       if(Parameters[i].Active!=0)
            N++;
//----------------Initialization of method-------------------------//
  if(Why==0)
   {
                NumberOfIterations=0;
                for(j=0; j<A.cols; j++)
                 {
                    for(i=0; i<A.rows; i++)
                              A.data[i][j] = Parameters[i].CurrentValue;          // problem in
                different combinations of Low & High.
                   Z= f(A, &Collide);
                 }
   }
        error=neldermead_one_iteration(Answer, CollisionFlag, &A, &Z, &Collide);
       NumberOfIterations++;

       if(NumberOfIterations>100 || error<=epsilon)
         return -1;  // Stop signal
        else
         return 1;
}
```

The second method is for the iteration process which is called in the first method itself. This method actually works with all the basic operations that nelder mead needs while finding the optimum value. Within this method all the essential points like Best point B, good point G, worse point W, Mean point M, Reflected point R, and conditional points E, C are evaluated. And then as the need arises search for the next better point is called.

The second method goes as follows (note all variables are defined in the code):

```
double neldermead_one_iteration(long *Answer, int *CollisionFlag, Matrix* A,
                                Matrix* Z, Matrix* Collide)
{
                alpha=1; gamma=2; beta=0.5;
```

```
//-----------------Determining Best Value-------------------//

        temp=best(get_vector(T(*Z), 0));
        Zb=temp.data[0][0];
        Y=temp.data[0][1];
        Ab=get_vector(*A,Y);

//-----------------Determining Second worse Value-----------//

        temp0=bad(get_vector(T(*Z), 0));
        Zg=temp0.data[0][0];
        X=temp0.data[0][1];
        Ag=get_vector(*A,X);

//-----------------Determining Worst Value------------------//

        temp1=worst(get_vector(T(*Z), 0));
        Zw=temp1.data[0][0];
        W=temp1.data[0][1];
        Aw=get_vector(*A,W);

//--------------------Mean--------------------------------//
        M = Mean(*A,W);
        Zm = f(M, &CollideM);

    //--------------------Reflection----------------------------//
        R = Reflection(M, Aw, alpha);
        Zr = f(R, &CollideR);

    //--------------------Process-----------------------------//
        if(Zr.data[0][0] < Zg)
         {
            if(Zb < Zr.data[0][0])
             {
                // Replace Aw point with R
                update(A, R, W);
             }
            else
             {
                // Compute Point E and its Z-value
                E = Expansion(R, M, gamma);
                Ze = f(E, &CollideE);
                //New Matrix A:
                if(Zb < Ze.data[0][0])
                 {
                    if(Ze.data[0][0] < Zg)
                        update(A, E, W);
```

```
                    else
                     {
                        if(Ze.data[0][0] < Zr.data[0][0])
                            update(A, E, W);
                        else
                            update(A, R, W);
                     }
                }
                else
                 {
                    if(Ze.data[0][0] < Zr.data[0][0])
                        update(A, E, W);
                    else
                        update(A, R, W);
                }
            }
        }
```

The process begins by finding the function values at point R and comparing with the function value at point G. This then asks the program to either replace the worse point with point R if *f*(R) is greater than *f*(G) or find point E and then replace W with point E or R as the condition arises. The "else" loop asks the program to Contract or Shrink the simplex and work to find better point there. And at each step the initial matrix A is updated with the correct column vector and finally the function value is evaluated and lastly best point and optimum value is found out.

The 'else' loop goes as follows:

```
//-------------------Contraction---------------------------//
        else
         {
            C = Contraction(Aw, R, M, beta);
            Zc = f(C, &CollideC);

            if(Zc.data[0][0] < Zg)
                update(A, C,W);
            else
             {
                if(Zr.data[0][0]<Zw)
                 {
                    update(A, R,W);
                    if(Zc.data[0][0]<Zw)
                        update(A, C,W);
```

```
                        if(Zc.data[0][0] == Zw)
                            Shrink_update(A, Ab);
                    }
                    else
                    {
                        if(Zc.data[0][0]<Zw)
                            update(A, C,W);
                        if(Zc.data[0][0] == Zw)
                            Shrink_update(A, Ab);
                    }
                }
            }
    *Z=f(*A, Collide);
    temp=best(get_vector(T(*Z), 0));
    Zb=temp.data[0][0];
    temp1=worst(get_vector(T(*Z), 0));
    Zw=temp1.data[0][0];

    return Abs(Zb-Zw);                    // error = |Zb-Zw|
}
```

This method finally returns the magnitude of error for further iteration. This error is compared with the user defined "epsilon" value which controls the number or accuracy up to which this simulation process is required to be carried on.

With this process code written, it was then the job to implement it in the PressOpt for the test run and further usage. For this the simulation code and simulationthread code and the OptPanel code were changed slightly by adding its caller function for simulation strokes.
Previously, answer was just equal to glRun.ProdRate. And optimized point was obtained by selecting maximum production rate, and in case of tie the answer with minimum mean accelerations was chosen.
Changes in **simulation.cpp**:

```
long SimulateOneStroke(int *CollisionFlag)
{
   ....
   ....
   .....
answer = -glRun.ProdRate + 0.6*glRun.MeanAccComp +0.2*glRun.MeanAccEmpty;

 return(answer);
}
```

Changes in **simulationthread.cpp**:

```
switch(glOptions.Method)
 {
  .........
  case 3:  // Neldermead
  maxprog=neldermead(0, &Answer, glParam, &collision);
  break;
 }

while(glRun.Busy==TRUE && status!=-1)
 {
   ......
   case 3:  // Neldermead
   status=neldermead(1, &Answer, glParam, &collision);
   break;
  }
```

Changes in **Optpanel.cpp**:

Note: As both “Max number of iteration” and “Error” were introduced, we can also introduce the parameters like alpha, beta, gamma, and the constants k1, k2 and k3 which are used in answer.

```
void __fastcall TMainForm::ComboBoxMethodChange(TObject *Sender)
 {
 glOptions.Method=ComboBoxMethod->ItemIndex;

 switch(glOptions.Method)
  {
   .......
   case 3: // NelderMead
    LabelParameter1->Caption="Max number of iteration";
    LabelParameter1->Visible=TRUE;
    EditParameter1->Visible=TRUE;

    LabelParameter2->Caption="Error";
    LabelParameter2->Visible=TRUE;
    EditParameter2->Visible=TRUE;
  break;
  }
```

Hence the more work to be done on this project are:

- Introducing a method for filling the Matrix A by using combination of Low and High. The idea that comes up is to use screening method initially and then perform the nelder mead method by filling the Matrix A.
- Debugging of a code that can create a file where the done simulations in the form of matrix and results can be saved for the simulations. Currently with the present code there appears to be some problem and hence requires correction.
- Since the two problems still exists, the TEST RUN for this optimization method though performed cannot be evaluated. Hence with this implementation lastly one needs to perform the Simulation and debug the code if any problem still exists. However, without this PressOpt version the test code for nelder mead ran successfully for the function corresponding to a “CIRLCE” giving its minimum value at its centre.
- The method to check previously performed simulations is yet to be introduced by getting the value of matrix A in some text file. Also hence, the other process of checking the previously done simulations too is left for implementation.

# 4. Inferences

## I. Results:

This project needs to perform few more TEST RUNS for Nelder Mead method to discuss and evaluate about any important results. While performing test run for Nelder Mead the result obtained in the text file is found to be insufficient to state about the success of the work. And hence one needs to debug the code if required by the results in the Test Run.

A test code for Nelder Mead process without any involvement of PressOpt was ran successfully before implementing in it. The results obtained were quite accurate when the optimization function was Circle, where one knows the minimum point would be at its centre. However there were few problems regarding the use of 'error = |Zb-Zw|' as if both these points somehow lies on the circumference of a circle then the process stops to work and says that as the optimum point. But apart from this, the process of optimization ran successfully, though in many steps.

## II. Advantages of Nelder Mead

Nelder mead method has few advantages when compared to the previously or yet to be implemented methods like Screening and Derivative. Nelder mead process of optimization has a specific path and direction in which it proceeds to find the optimum value, while, screening method has no such direction. Screening method performs all strokes in simulation and then compares all its result to give the best output. Hence though it might find the best answer at the first stroke it would continue its process unless it checks all the obtained combinations. A two full factorial Screening method checks for $2^n$ combinations for 'n' active parameters, hence when there exists large number of active parameters, the process takes days to complete the simulation. While, the Nelder Mead method requires to check comparatively less number of points to reach the optimum solution (when n is large enough).

Nelder Mead method since does not involve derivatives of the defined function for optimization it faces no problem in solving a discontinuous function. Hence when pressopt faces a collision point nelder mead method can define that point as one of the point of discontinuity and can work ahead. However in derivative method one can not look and jump over points of collision. And hence derivative method needs to know its boundaries defined by collision point so has to work within that space.

## III. Some Problems with Nelder Mead:

Nelder Mead method faces one major disadvantage that the process might turn too lengthy and time consuming if the optimum point is far away from the initially defined Points to start with. Also when compared to the Screening method, the number of iterations performed by nelder mead when the number of active parameters are less is much large. By controlling the number of iterations accuracy is affected (when the iteration number is less). Also nelder mead method only deals with the situation when the required function needs to be minimized. Hence weighted constants are introduced, sometimes unwontedly, to use this method.

## IV. Future Aspects and Other Methods:

With having knowledge of advantages and disadvantages, we can try a combination of all the so far done methods to obtain better results.

Hence, if possible, a combination of Nelder Mead method with screening and derivative methods can help us to provide a faster way to reach an optimum point. For example: performing screening method first can let us know what should be the starting points of nelder mead method could be so that the results obtained are fast and accurate. (No comment about the derivative method since this is yet to be implemented. But one can think of an idea where derivative method uses nelder mead method at the point of collision and hence would know the direction in which the derivatives can help.)

# 5. Acknowledgement

I am very thankful to University West, Trollhattan Sweden, for inviting me to work as a Summer Intern to the project of Optimization of Complex automated production line for Volvo Manufacturing Unit, under Dr. Fredrik Danielsson and Mr. Bo Svensson.

Also I am very thankful to my guides Dr. Fredrik Danielsson and Mr. Bo Svensson who had helped me at each and every stage of this project by giving their valuable time in clearing my questions regarding the work procedure and difficulties that I had faced in doing so. They both had helped me in understanding each and every step necessary for doing this project.

I am also thankful to T. C. Kasongo (Cyril) who had helped me in understanding the basic code structure of PressOpt, making me familiar with each and every elements of importance related to it. Cyril has also provided me with his helping hand in clearing few of the obstacles faced after his project completion too and also has also provided me with all the resources he had.

Thank you,

Yogesh Jain.

# 6. References


- **A Virtual Real Time Model for Control Software Development – Applied on a Sheet-Metal Press Line**. – Bo Svensson, Fredrik Danielsson and Bengt Lennartson.
- **Off-line Optimization of Complex Automated Production Lines – Applied on a Sheet-Metal Press Line.** – Bo Svensson, Fredrik Danielsson and Bengt Lennartson.
- **Distributed System Architecture for Optimizing Control Logic in Complex manufacturing Systems.** – Fredrik Danielsson
- **40 years of the Nelder-Mead Algorithm.** – David Byatt, Ian Coope, Chris Price.