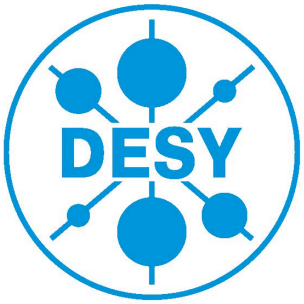



# Java and Matlab implementation of Beam Based Alignment method for the control system virtual XFEL servers

Miguel Camacho Aguilar, University of Seville, Spain

Supervised by Dr. Pedro Castro-García, Machine Physics Group, DESY

September 9, 2014

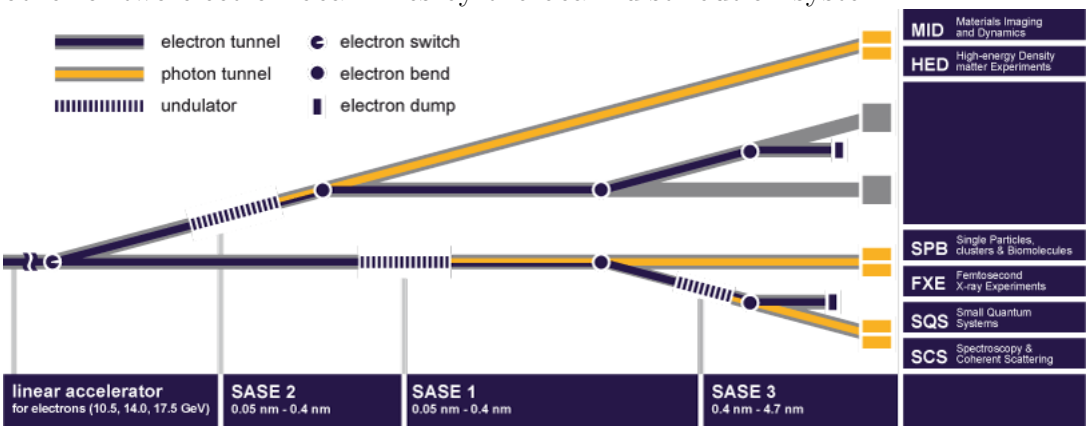
**Abstract**

The beam based alignment method developed in [3] was implemented and tested for the control system of the future European XFEL using Java classes and MATLAB scripts working together to collect data from the simulations and calculate the misalignments of the SASE1 section devices. It was also made flexible to align any set of quadrupoles and BPMs from the layout in case that some of them do not work properly. Results from the alignment using random initial conditions are also presented.

# Contents

# 1. Introduction

The European X-Ray Free Electron Laser (from now European XFEL) will be the largest X-Ray laser facility in the world with a 3.4 km long tunnel that begins on the DESY site in Hamburg-Bahrenfeld and runs to the XFEL research site, which is to be erected south of the town of Schenefeld.
The basic functions of the main components are schematically described in the following: In the injector, electron bunches are extracted from a solid cathode by a laser beam, focused, then accelerated by an electron radio frecuency gun and directed towards the linac with an exit energy of 120 MeV. Due to the small mass of the electrons (0.5 MeV), from this point we can assume the ultrarelativistic limit, where $\beta \approx 1$.
In the linac, consisting of a 1.6 km long sequence of superconducting accelerating modules, magnets for beam steering and focusing, and diagnostic equipment, the electrons can be accelerated to energies of up to 20 GeV. The reason why superconducting radiofrecuency cavities are used is their ultra-low electrical resistivity that allows store energy with a very low loss and narrow bandwidth.
At the end of the linac, the individual electron bunches are channelled down one or the other of two electron beamlines by the beam distribution system.




After that, electron bunches are conducted through the undulators where they will produce, depending on the beamline, different kinds of X-Ray photons from 0.05 nm to 4.7 nm wavelengths. Those undulators will have lengths exceeding 100 m (SASE3) and even 200 m (SASE1 and SASE2) and very tight tolerances for alignment and uniformity due to the need of a perfectly coherent emission. In order to achieve those requirements undulators will be splitted in 5 m modules, filling the spaces with focusing quadrupoles and diagnostic equipment such as Beam Position Monitors. Small misalignments on these quadrupoles, which are needed to focus the beam along the undulators, introduce distortions on the electron beam trajectory which can disrupt the overlap between the photon field and the electron bunch. Therefore, quadrupoles are mounted on micromovers that will allow to correct the beam orbit once misalignments are known. To calculate those misalignments we will apply beam-based alignment techniques.

# 2. Charge Beam Dynamics

Similar as in light optics, the change in position and angle of a single charged particle from one point of the z axis to another is calculated through a transfer matrix $R_{ij}$.

$$\begin{bmatrix} x \\ x' \end{bmatrix}_{z_2} = \begin{bmatrix} R_{11} & R_{12} \\ R_{21} & R_{22} \end{bmatrix}_{z_1 \to z_2} \begin{bmatrix} x \\ x' \end{bmatrix}_{z_1} \tag{1}$$

This marix can be expressed in terms of Courant-Snyder parameters. Using [4] we know that the transfer matrix between $z_1$ and $z_2$ is:

$$R = \begin{bmatrix} \sqrt{\frac{\beta_2}{\beta_1}}(\cos\phi + \alpha_1 \sin\phi) & \sqrt{\beta_1\beta_2}\sin\phi \\ \frac{\alpha_1 - \alpha_2}{\sqrt{\beta_1\beta_2}}\cos\phi - \frac{1+\alpha_1\alpha_2}{\sqrt{\beta_1\beta_2}}\sin\phi & \sqrt{\frac{\beta_1}{\beta_2}}(\cos\phi - \alpha_2\sin\phi) \end{bmatrix} \tag{2}$$

Where $\phi = \mu_2 - \mu_1$, and ($\alpha_1$, $\beta_1$,$\mu_1$) , ($\alpha_2$, $\beta_2$,$\mu_2$) are the values of the Courant-Snyder parameters [2] for $z = z_1$ and $z = z_2$ respectively. Using this we will be able to relate the variations in the orbit of the beam to the launch parameters.

## 2.1. Calculating the effect of quadrupole displacement

As first step, we calculate the effect on the measure with a certain beam position monitor i due to the misalignment $d$ of another certain quadrupole j.
Let $Q_{ij}$ be the transfer matrix associated to a quadrupole in his own reference system $\bar{x}, \bar{y}, \bar{z}$ such that

$$\begin{bmatrix} \bar{x} \\ \bar{x'} \end{bmatrix}_2 = \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \begin{bmatrix} \bar{x} \\ \bar{x'} \end{bmatrix}_1 \tag{3}$$

Actually [4], for a focusing quadrupole ($k > 0$):

$$Q = \begin{bmatrix} \cos\sqrt{k}l & \frac{1}{\sqrt{k}}\sin\sqrt{k}l \\ -\sqrt{k}\sin\sqrt{k}l & \cos\sqrt{k}l \end{bmatrix} \tag{4}$$

Or, in case of defocusing quadrupole ($k < 0$):

$$Q = \begin{bmatrix} \cosh\sqrt{-k}l & \frac{1}{\sqrt{-k}}\sinh\sqrt{-k}l \\ \sqrt{-k}\sinh\sqrt{-k}l & \cosh\sqrt{-k}l \end{bmatrix} \tag{5}$$

Where k is the quadrupole strength defined as follows:

$$k = \frac{e}{\beta E} g \tag{6}$$

And e is the electron charge, E is the energy of the beam and g is the gradient of the magnetic field inside the quadrupole, defined as $g = \frac{\partial B_x}{\partial y} = \frac{\partial B_y}{\partial x}$.
Now, let us suppose that the quadrupole is displaced a distance $d$ along the x axis. Changing the reference system we can calculate the new transfer matrix:

$$\begin{cases} \bar{x_1} = x_1 - d \\ \bar{x_1'} = x_1' \end{cases} \tag{7}$$

Using this, we obtain:

$$\begin{bmatrix} x_2 - d \\ x_2' \end{bmatrix} = \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \begin{bmatrix} x_1 - d \\ x_1' \end{bmatrix} \tag{8}$$

From now ($x_1$,$x_1'$) will always represent the position and the angle with respect to the x axis just before the studied quadrupole.

$$\begin{bmatrix} x_2 \\ x_2' \end{bmatrix} = \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_1' \end{bmatrix} + d \begin{bmatrix} 1 - Q_{11} \\ -Q_{22} \end{bmatrix} \tag{9}$$

After a certain optical system, represented by the transfer matrix $R_{ij}$, we can relate the variation of the position of the beam with respect to the original trajectory.

$$\begin{bmatrix} x \\ x' \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} \\ R_{21} & R_{22} \end{bmatrix} \begin{bmatrix} x_2 \\ x_2' \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} \\ R_{21} & R_{22} \end{bmatrix} \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_1' \end{bmatrix} + \begin{bmatrix} R_{11} & R_{12} \\ R_{21} & R_{22} \end{bmatrix} d \begin{bmatrix} 1 - Q_{11} \\ -Q_{22} \end{bmatrix} \tag{10}$$

We can only measure $x$, that we obtain using the previous equation

$$x = R_{11}(Q_{11}x_1 + Q_{12}x_1' + d(1 - Q11)) + R_{12}(Q_{21}x_1 + Q_{22}x_1' - dQ_{21}) \tag{11}$$

And finally we obtain an expression for the change of the orbit due to the misalignment of the quadrupole

$$\Delta x = d(R_{11}(1 - Q_{11}) - R_{12}Q_{21}) \tag{12}$$

Since the displacement of the orbit is proportional to the misalignment of the quadrupole, and because of the linearity of the problem, we can add the misalignments of the other quadrupoles. Then we define the response matrix $M_{ij}$:

$$M_{ij} = R_{11}^{ij}(1 - Q_{11}^{j}) - R_{12}^{ij}Q_{21}^{j} \tag{13}$$

That is, the displacement of the orbit at the $i^{th}$ BPM due to the misalignment of the $j^{th}$ quadrupole. If N is the number of BPMs and M is the number of quadrupoles, in matrix form it is as follows

$$\begin{bmatrix} \Delta x_1 \\ \Delta x_2 \\ \vdots \\ \Delta x_N \end{bmatrix} = \begin{bmatrix} M_{11} & \dots & M_{1M} \\ \vdots & \ddots & \vdots \\ M_{N1} & \dots & M_{NM} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_M \end{bmatrix} \tag{14}$$

Where $\Delta x_i$ is the displacement of the beam trajectory at the $i^{th}$ BPM due *only* to the misalignment of the quads.

It is important to remark that those transfer matrices will depend on the energy of the beam, as we know that the curvature radio of the trajectory of the beam passing through a dipole or a quadrupole depends on its energy. Therefore, for each energy of the beam we will have to calculate those matrices. An illustrative example is to set the first quadrupole misalignment to one millimeter, and calculate the trajectory.

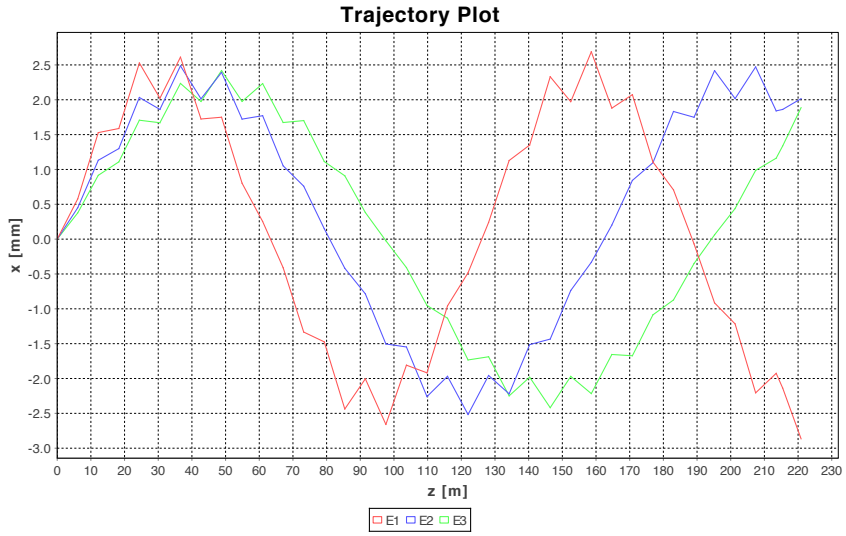



Figure 1: XZ projection of the trajectory of the beam for a one millimeter displacement of the first quadrupole in section SASE1 for energies: E1=8.0 GeV, E2=14.0 GeV, and E3=17.5 GeV

In fig. 1 we can see the different behavior when we increase the energy of the beam.

## 2.2. The Launch Matrix

It is easy to see that if we want to study the alignment of a certain section of the beam line we will need to let the initial position and angle of the beam as unknowns, because they will depend on many devices between the start of the accelerator and our SASE section. As we would expect the beam to be coincident with the z axis of the accelerator ($x_0 = x'_0 = 0$) the deviation from the requested parameters will be just the actual values of the launch parameters.
In this case, the problem is straight forward because we only need the transfer matrices between the start point of the section and each BPM position where $L_{ij} = R_{ij}$, and $R_{ij}$ is given in eq.( 2)

$$\begin{bmatrix} \Delta x_1 \\ \Delta x_2 \\ \vdots \\ \Delta x_N \end{bmatrix} = \begin{bmatrix} L_{11} & L_{12} \\ \vdots & \vdots \\ L_{N1} & L_{N2} \end{bmatrix} \begin{bmatrix} x_0 \\ x'_0 \end{bmatrix} \tag{15}$$

## 2.3. Misalignment of the beam position monitors

There is still one contribution left to the measurements of the monitors, and it is that they can be also misaligned, so we only need to do a change of reference system:

$$\Delta x_i = -\beta_i \tag{16}$$

In addition, there will be also electronic noise that will not allow us to have more than a certain precision. This noise will have a random component, so the only way to reduce it is to measure more than once and use the mean value of the measurements.

## 2.4. Beam Based Alignment Method

If we sum all the contributions (we can do it because our problem is linear) we end up with the following system of equations:

$$\begin{bmatrix} \Delta x_1 \\ \Delta x_2 \\ \vdots \\ \Delta x_N \end{bmatrix} = \begin{bmatrix} L_{11} & L_{12} \\ \vdots & \vdots \\ L_{N1} & L_{N2} \end{bmatrix} \begin{bmatrix} x_0 \\ x_0' \end{bmatrix} + \begin{bmatrix} M_{11} & \dots & M_{1M} \\ \vdots & \ddots & \vdots \\ M_{N1} & \dots & M_{NM} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_M \end{bmatrix} - \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_N \end{bmatrix} + \begin{bmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_N \end{bmatrix} \tag{17}$$

Where the last column-matrix is the one associated to the uncertainty of the measurements due to instrumental limitations or electronic noise. Everything done here can be applied to the problem in the YZ plane, as we will do later.
We can see that the total amount of unknowns are M quadrupole misalignments, N BPM misalignments and 2 launch parameters. Since we need to set a reference system, we can choose two misalignments to be zero so we obtain a reference z axis. However, the number of equations is only N since we only have N BPM measurements.
The system has more unknowns than equations so there is an infinite number of valid solutions, and that would not allow us to get any information. In order to obtain more equations we repeat the measurements for more than one energy without moving any monitor or quadrupole. Then we will have:

$$N_{unknowns} = N + M + 2N_{Energies} - 2 \tag{18}$$

$$N_{equations} = N_{Energies} N \tag{19}$$

We need, at least, the same number of equations than unknowns, so the number of energies that we need will be:

$$N_{Energies} \geq \frac{N + M - 2}{N - 2} \tag{20}$$

In SASE1, there are 38 BPMs and 36 quadrupoles so only 2 energies would be needed. However, the statistical noise of the measurements will introduce errors in the solution.

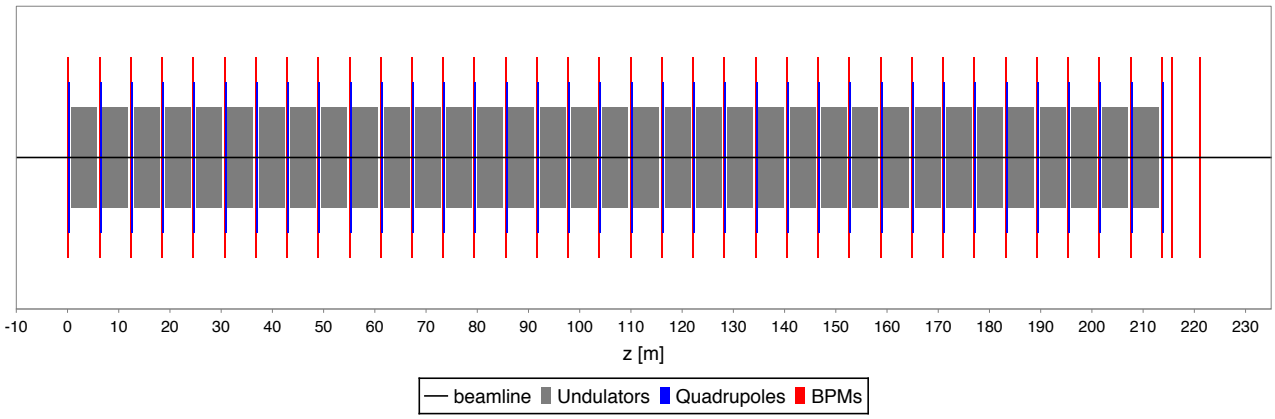



Figure 2: Schematic plot of the SASE1 section layout

in order to reduce the noise dependency in the system of equations we will take measurements for at least three different energies. This is the main idea of the method that we will apply.
A plot of the layout of SASE1 section and a plot of a cell are showed in fig. 2 and fig. 3 respectively.

## 2.5. Single Value Decomposition

To solve the system of equations, we use the Single Value Decomposition method (from now SVD), where the coefficient matrix is decomposed in the product of three matrices: one diagonal matrix and two orthogonal matrices. Using this method we will be able to find a solution even if we have less equations than unknowns, minimizing the module of the solution-vector. Let be a system of equations such that:

$$\mathbf{b} = \mathbf{A}\mathbf{x} \tag{21}$$

Where $\mathbf{b}$ is the vector containing the readings of the BPMs, $\mathbf{A}$ is the matrix of the coefficients that relate the misalignments of BPMs, quadrupoles and the initial launch parameters to the value of the position of the beam at each BPM given in eq. (17) and $\mathbf{x}$ is the vector containing the unknowns: BPMs and quadrupole misalignments and launch parameters.
Then, using the SVD theorem, we can write A as follows:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \tag{22}$$

Where $\Sigma$ is a diagonal matrix containing the singular values of the matrix A, with as many non-zero values as the range of the matrix A, U and V are orthogonal matrices that satisfy the following: $U^{-1} = U^T$.

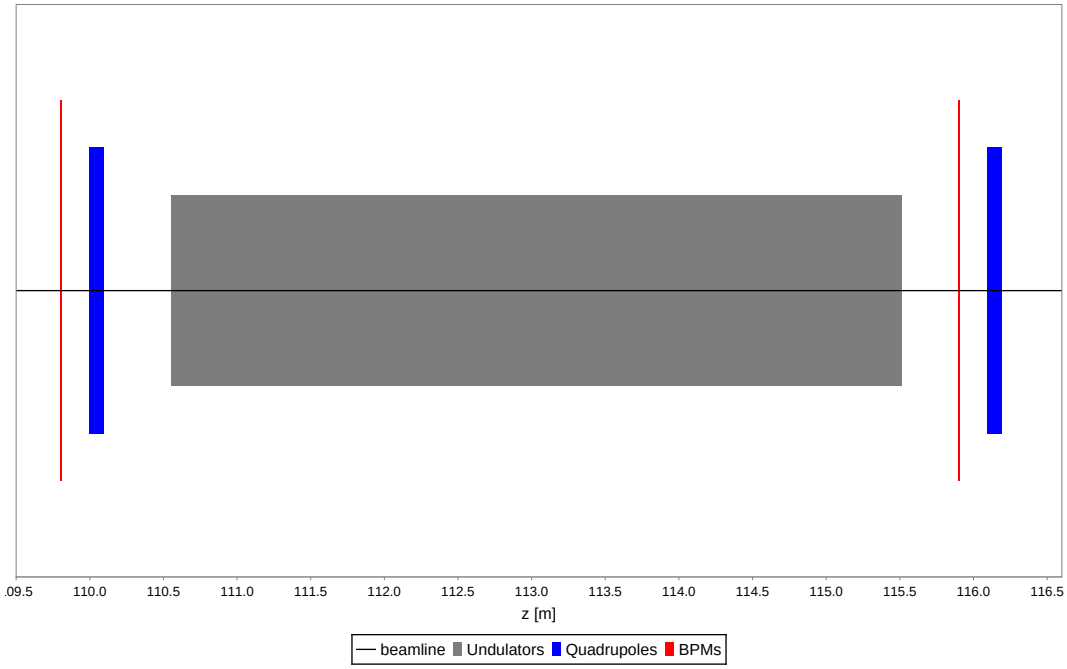



Figure 3: Schematic plot of the SASE1 cell layout

Then, the solution will be given by the following expression.

$$\mathbf{x} = \mathbf{V} \begin{bmatrix} \sigma_1^{-1} & & & & \\ & \sigma_2^{-1} & & & \\ & & \sigma_3^{-1} & & \\ & & & \ddots & \\ & & & & \sigma_N^{-1} \end{bmatrix} \mathbf{U}^T \mathbf{y} \tag{23}$$

Where only non-zero single values are taken into account. Zero single values are discarded eliminating one equation of the system for each of them. Furthermore, this theorem let us know how many independent equations we have, by giving us as many non-zero single values as the number of independent equations.
However, due to the random noise of the readings of the BPMs, we can get singular values that are very close to zero but not zero. In that case, we define a tolerance close to zero, below which we consider the singular value equal to zero. This way we discard one equation for each singular value below this tolerance and we do not use equations that would add non-physical constraints to the solution. In case that we end up with less equations than unknowns, this method will provide us the solution that minimizes the quadratic sum of the unknowns.

# 3. Implementation of the BBA Method

The BBA procedure consists on:

- Calculate the response matrix $M_{ij}$ and the launch matrix $L_{ij}$ for several energies.
- Obtain the readings from the BPMs for each energy.
- Solve the overdetermined system of equations.
- Correct the position of the quadrupoles, repeat the measurements and solve again until the trajectory satisfies our tolerance.

In preparation for the future XFEL control system, the beam-based alignment algorithm has been adapted to connect to the virtual XFEL control system. Thus, we have a set of servers that will behave as the real ones:

- OpticsServer, where all the information of the accelerator is stored. It can be accessed using MATLAB and a large set of parameters (like the position and strength of quadrupoles, the energy of the beam) can be modified. It is integrated with ELEGANT [1], which is a simulation code written for APS at Argonne Laboratories that is used to run the simulations
- Beam Position Monitors server, where their measurements are collected from the BPM electronics, averaged and stored, and their values can be accessed using MATLAB or JAVA.

The results of the ELEGANT beam trajectory simulations can be transferred to the BPM server, where some random noise is added. Then, the BPM readings can be accessed from our JAVA classes in order to solve the problem.

## 3.1. Calculating the Matrices

### 3.1.1. Response Matrices

We use three different methods to obtain the response matrix $M_{ij}$.

**The first one** is using ELEGANT to simulate the trajectory for an unitary displacement of only one quadrupole while all the others are perfectly aligned. The resulting beam position at each BPM corresponds to the element of the matrix $M_{ij}$. This method requires 36 (number of quadrupoles in SASE1) simulations for each energy and for each axis (x and y), so it is a very slow method. For this method we need to have the layout file of the accelerator.

**The second one** uses the eq. (13) in section 2.1, where we used the transfer matrix from the quadrupole to the BPM and the original transfer matrix of the quadrupole. All needed parameters can be read from the OpticsServer by running only one simulation for each energy , so this method is much faster than the first one.
MATLAB scripts were written for both methods, implemented and tested, obtaining the same results for all the values of energy. The main difference is the fact that the server needs a long time (about 4 s) to recalculate all the parameters becoming a big source of difficulties when first method is used.

### 3.1.2. Launch Matrices

To obtain these matrices we calculate the matrix R as we did in the previous section, using the values of the Courtant-Snyder parameters just before SASE1 and the values at the BPMs.
In addition, those scripts were written to save all the matrices in files in order to make easier the interaction with JAVA classes.

## 3.2. Solving the equations

We wrote a JAVA class implementing the method described in section 2.5. However, it makes the computer calculate many products that involve zeros, see eq. (23). Because of that, we simplified the computation in the following way:

$$\mathbf{x} = \left[ \sum_{i=1}^{N} \sigma_i^{-1} \mathbf{v}_i \mathbf{u}_i^T \right] \mathbf{y} \tag{24}$$

Using this simplified method we obtained a reduction in the computing time.
In order to make the class as flexible as possible, we added some features:

- It will accept any number of quadrupoles and BPMs, and any given layout, and will calculate the matrices associated to that layout.
- It will allow us to choose any number of energies and any values for them. Therefore we will be able to use more equations and reduce the effect due to the noise.
- It will give us the possibility to choose which quadrupoles and BPMs from the layout we want to use, preventing the possibility of that some of them do not work or they are disconnected.

To solve the problem we have to follow the procedure:

- Set the layout, the number of quadrupoles, BPMs that we want to use and energies.
- Set the response and launch matrices for each energy. For this we need the scripts from the previous section. As they were written in MATLAB, now we need another JAVA class to read the matrices from the files.
- Solve the problem and get the values of the displacements of the quadrupoles, BPMs and the launch parameters for each energy.

## 3.3. Corrections

Once we have obtained the displacements of the quadrupoles and BPMs and the launch parameters, we modify the positions in the OpticsServer, where the trajectory will be recalculated. After that, using MATLAB we will upload that trajectory to the BPM server, and it will add random noise to simulate electronic noise. Actually, launch parameters can not be modified because they are a consequence of previous sections of the accelerator.
The scheme of the interactions between servers and written programs is showed in fig. 4. There it can seen how MATLAB Scripts and JAVA classes interact through temporal files where all the information is stored, so we can use it more than once without having to connect to the server again.

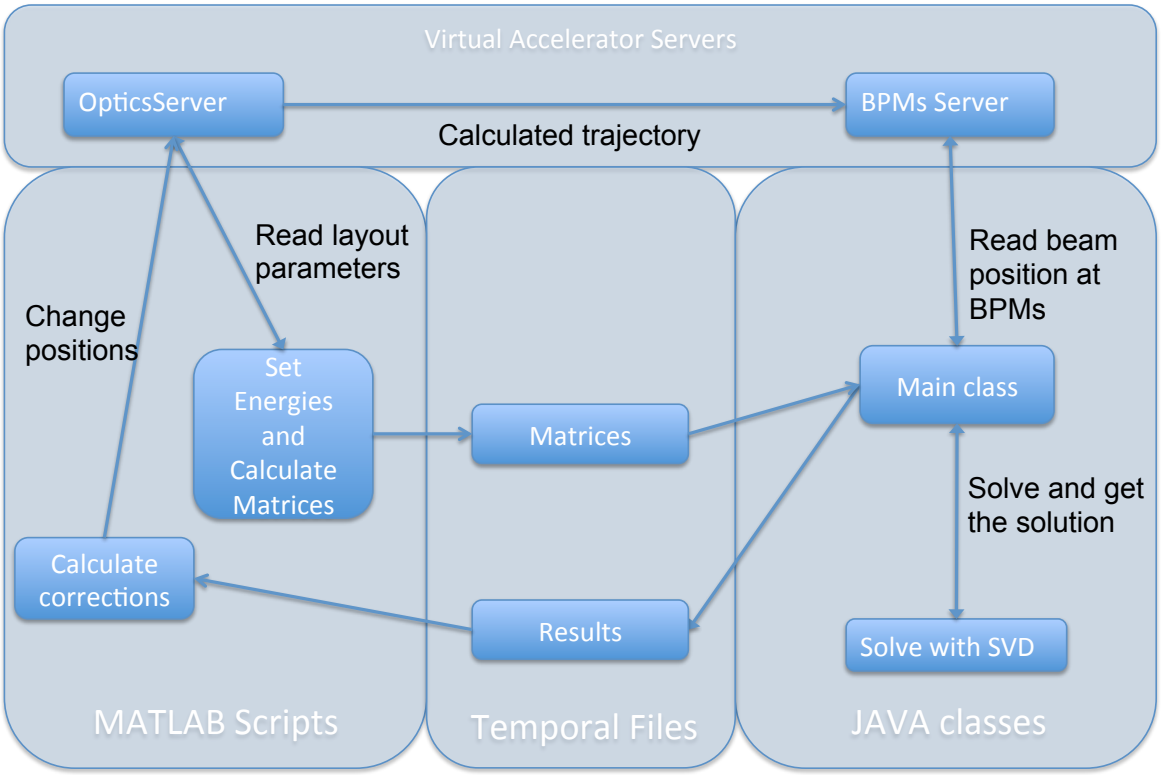



Figure 4: Scheme of connections between MATLAB scripts, Java classes and virtual accelerator servers

# 4. Results

We have applied this method to a simulation where we have included a set of random initial parameters:

| Errors (Gaussian rms) | Value | Unit |
|---|---|---|
| Quad. rms misalignment | 0.3 | $mm$ |
| BPM rms misalignment | 0.3 | $mm$ |
| BPM rms resolution | 1 | $\mu m$ |
| Launch position rms | 0.1 | $mm$ |
| Launch angle rms | 0.1 | $mrad$ |

Table 1: List of errors used in simulations. Each error has a gaussian distribution

As we discussed in section 3.2, when we add random noise to the simulated readings of the BPMs, we obtain a larger number of non-zero singular values when SVD is applied. Those non-zero values would lead us to a non-physical solution due to the fact that we are using equations that actually are not independent. To reduce this effect, we set a relative tolerance of about $10^{-5}$, that is, if a singular value is less than the biggest one times this relative tolerance, we will consider it equal to zero and we will discard one equation. Using this tolerance, we had to discard one equation, obtaining a much better result than setting the tolerance equal to zero, as we explained before.
In fig. 5 we can see the improvement in the trajectory, for an energy of 7 GeV, due to the corrections made only to the quadrupole positions (blue line) and the trajectory that we would achieve if we could modify the launch parameters (green line). The original trajectory had a RMS of 2.47 mm while the one with the corrections of the quadrupoles has a RMS value of 0.52 mm and the one with corrections to launch parameters has a value of 0.02 mm.
In fig. 6 we can see the improvement in the alignment of the quadrupoles after applying the method. Those remaining misalignments are due to the electronic noise of the BPMs simulated measurements, so the best procedure would be to increase the number of measurements and use the average.

# 5. Conclusions and possible improvements

We have tested and implemented the method used in [3] in the new virtual environment of the European XFEL control system developed by DESY. The results obtained and presented in section 4 are very similar to the results presented in [3], therefore proving that the implementation is correct and valid for our purposes. We have tested several calculation methods and we have made our classes to accept any set of BPMs and quadrupoles from the layout, what is really useful in case that they stop working, as it can happen in a real accelerator. It can be also applied to other sections such as SASE2 or SASE3. The main future improvements are the iterative procedure that consists

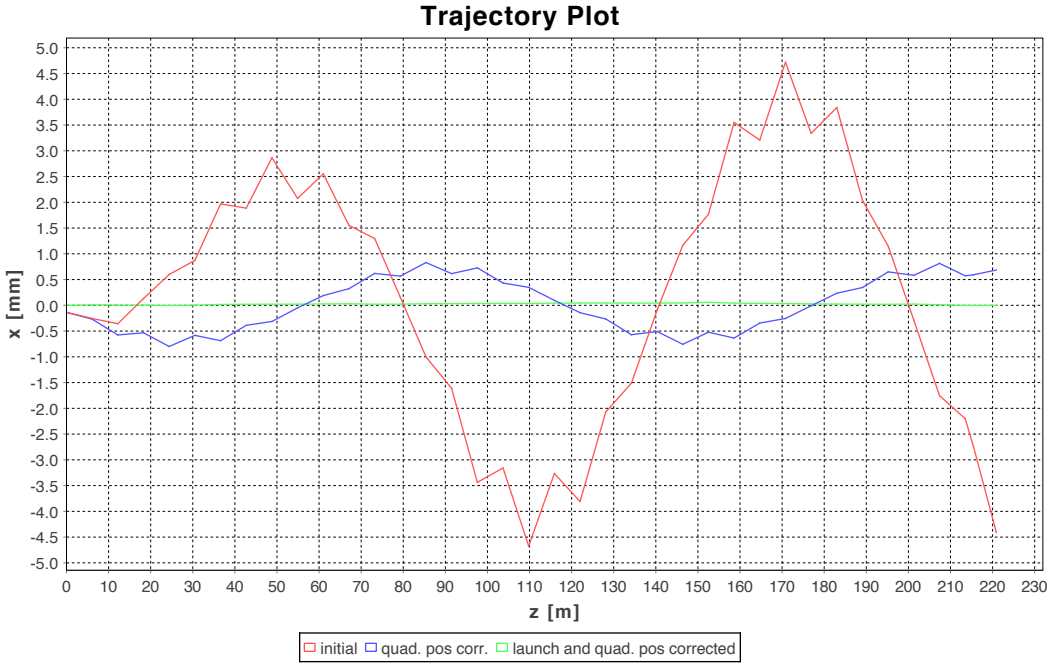



Figure 5: Trajectory of the beam before and after corrections

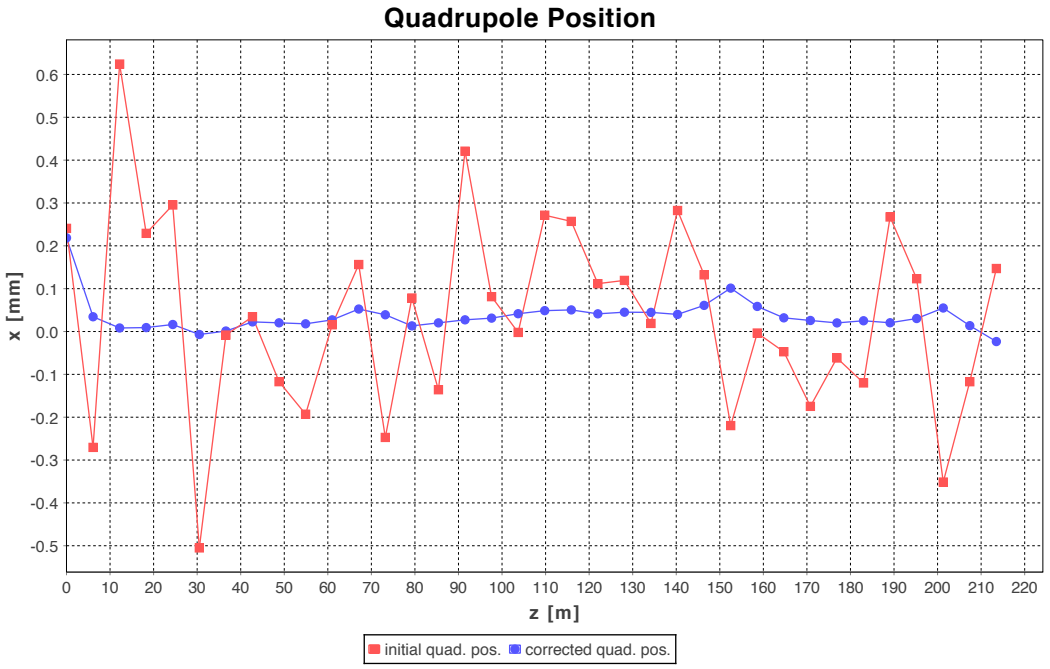



Figure 6: Quadrupole positions before and after the appliance of the method

on applying this method until a stable solution is reached as done in [3] and the final implementation with the real accelerator. Another improvement could be to program the entire method in JAVA classes, that could give a better results in terms of data management and flexibility.

# A. Appendix: Java Class for Solving Using SVD

This is the class written by us to solve the equations:

```
  package xfelbbasolver;
import org.apache.commons.math.linear.Array2DRowRealMatrix;
import org.apache.commons.math.linear.DecompositionSolver;
import org.apache.commons.math.linear.QRDecompositionImpl;
import org.apache.commons.math.linear.RealMatrix;
import org.apache.commons.math.linear.SingularValueDecompositionImpl;
import org.apache.commons.math.stat.StatUtils;

public class SolveUsingSVD {
  public double singularValueNormalizedTolerance = 1.0e-9;
  private int numberOfEnergies = 3;
  private int numberOfBPMS = 37;
  private int numberOfQuads = 35;
  private int numberOfBPMOffsets = numberOfBPMS - 2;
  private double[][][] responseMatrixQuad;  //   [ index-Energy ] [ index-BPM ] [ index-Quad ]
  private double[][][] matrixLaunch;        //   [ index-Energy ] [ index-BPM ] [ 0:x  or 1:xp ]
  private double[] orbitError;
  private RealMatrix matV;
  private RealMatrix matU;
  private RealMatrix totalMatrix;
  private int numberOfIndenpendentEquations;

  private double[] singularValues;
  private double[][] xxp_final;
  private double[] misLaunch;
  private double[] misQuad;
  private double[] misBPM;

  public static SolveUsingSVD getSolver(MeasureAndOptics[] arrayMeasureAndOptics) {
    if ( (null == arrayMeasureAndOptics) || (arrayMeasureAndOptics.length < 2) ) {
      return null;
    }
    SolveUsingSVD solveUsingSVD = new SolveUsingSVD();
    solveUsingSVD.setNumberOfEnergies( arrayMeasureAndOptics.length );
    solveUsingSVD.setNumberOfBPMs(     arrayMeasureAndOptics[0].responseMatrixQuad.length);
    solveUsingSVD.setNumberOfQUADs(    arrayMeasureAndOptics[0].responseMatrixQuad[0].length);

    for(int iEnergy=0; iEnergy<arrayMeasureAndOptics.length; ++iEnergy) {
      solveUsingSVD.setLaunchMatrixForEnergy  (1+iEnergy, arrayMeasureAndOptics[iEnergy].launchMatrix);
      solveUsingSVD.setResponseMatrixForEnergy(1+iEnergy, arrayMeasureAndOptics[iEnergy].responseMatrixQuad);
      if (null != arrayMeasureAndOptics[iEnergy].posMeasurements) {
        solveUsingSVD.setOrbitErrorforEnergy    (1+iEnergy, arrayMeasureAndOptics[iEnergy].posMeasurements);
      }
    }
    return solveUsingSVD;
  }

  public SolveUsingSVD() {
    initializeArrays();
  }

  private void initializeArrays() {
    matrixLaunch      = new double[numberOfEnergies][numberOfBPMS][2];
    responseMatrixQuad = new double[numberOfEnergies][numberOfBPMS][numberOfQuads];
    orbitError = new double[ numberOfEnergies*numberOfBPMS ];
  }

  public void setNumberOfBPMs(int i) {
    numberOfBPMS = i;
    numberOfBPMOffsets = numberOfBPMS - 2;
    initializeArrays();
  }

  public void setNumberOfQUADs(int i) {
    numberOfQuads = i;
    initializeArrays();
  }
```

```
public void setNumberOfEnergies(int i) {
  numberOfEnergies = i;
  initializeArrays();
}
public void setOrbitErrorforEnergy(int i, double[] posMeasurements) {
  System.arraycopy(posMeasurements, 0, orbitError, (i-1)*numberOfBPMS, numberOfBPMS);
}
public void setResponseMatrixForEnergy(int indexEnergy, double[][] mat) {
  if (mat == null) {
    System.err.println("error:  mat = null");
    return;
  }
  if (mat.length != numberOfBPMS) {
    System.err.println("error:  mat.length != number_Of_BPMS");
    return;
  }
  if (mat[0].length != numberOfQuads) {
    System.err.println("error:  mat.length != 2");
    return;
  }
  for(int i=0;i<numberOfBPMS;i++){
    for(int j=0;j<numberOfQuads;j++) {
      responseMatrixQuad[indexEnergy-1][i][j] = mat[i][j];
    }
  }
}
public void setLaunchMatrixForEnergy(int indexEnergy, double[][] mat) {
  if (mat == null) {
    System.err.println("error:  mat = null");
    return;
  }
  if (mat.length != numberOfBPMS) {
    System.err.println("error:  mat.length != number_Of_BPMS");
    return;
  }
  if (mat[0].length != 2) {
    System.err.println("error:  mat.length != 2");
    return;
  }
  for(int i=0;i<numberOfBPMS;i++){
    for(int j=0;j<2;j++) {
      matrixLaunch[indexEnergy-1][i][j] = mat[i][j];
    }
  }
}
public void solveAndMeasureTime(int method) {
  long time_start, time_end;
  time_start = System.nanoTime();
  solve(method);
  time_end = System.nanoTime();
  System.out.println("the task has taken " + ( time_end - time_start )           + " nanoseconds");
  System.out.println("the task has taken " + ( time_end - time_start )/1000      + " microseconds");
  System.out.println("the task has taken " + ( time_end - time_start )/1000/1000 + " milliseconds");
}
public void solve(int method) {
  switch(method) {
    case 1: solveUsingInverseMatrix(); break;
    case 2: solveUsingPseudoInverseMatrix(); break;
    case 3: solveUsingSimplification(); break;
  }
}
public void constructMatrix() {
  double[][] matrix = new double[numberOfEnergies*numberOfBPMS][numberOfBPMS+numberOfQuads+2*numberOfEnergies];
  for(int i=0;i<numberOfEnergies*numberOfBPMS;i++) {
    for(int j=0;j<numberOfBPMS+numberOfQuads+2*numberOfEnergies;j++){
    matrix[i][j]=0;
    }
  }
  for(int i=0;i<numberOfBPMS;i++) {
    for(int j=0;j<numberOfQuads;j++) {
      for(int e=0;e<numberOfEnergies;e++) {
        matrix[i+e*numberOfBPMS  ][j]=responseMatrixQuad[e][i][j];
      }
    }
  }
  for(int i=0;i<numberOfBPMS;i++) {
    for(int j=numberOfQuads;j<numberOfQuads+numberOfBPMS;j++){
```

```
        if (j-numberOfQuads==i) {
          for(int e=0;e<numberOfEnergies;e++) {
            matrix[i+e*numberOfBPMS][j]=-1;
          }
        }
      }
    }
    for(int i=0;i<numberOfBPMS;i++) {
      for(int j=numberOfQuads+numberOfBPMS;j<numberOfQuads+numberOfBPMS+2;j++) {
        for(int e=0;e<numberOfEnergies;e++) {
          matrix[i+e*numberOfBPMS][j+2*e] = matrixLaunch[e][i][j-numberOfBPMS-numberOfQuads];
        }
      }
    }
    //Selection of desired columns, due to two freedom degrees.
    //We skip the first and last BPM.
    totalMatrix = new Array2DRowRealMatrix(matrix);
    int[] rows = new int[totalMatrix.getRowDimension()];
    int[] cols = new int[totalMatrix.getColumnDimension() - 2];
    for(int i=0;i<rows.length;i++){
      rows[i]=i;
    }
    int indexShift = 0;
    for(int j=0;j<cols.length;j++){
      if (j==numberOfQuads) {
        ++indexShift;
      }
      if (j==(numberOfQuads+numberOfBPMS-2)) {
        ++indexShift;
      }
      cols[j]=j + indexShift;
      //System.out.println(" j = " + j + "  cols[j] = " + cols[j]);
    }
    totalMatrix = (Array2DRowRealMatrix)totalMatrix.getSubMatrix(rows, cols);
  }
  public void svdDecomposition() {
    constructMatrix();
    SingularValueDecompositionImpl singularValueDecompositionImpl = new SingularValueDecompositionImpl(totalMatrix);
    singularValues = singularValueDecompositionImpl.getSingularValues();
//    for(int i=0; i<singularValues.length; ++i) {
//      System.out.println("s(" + (i+1) + ")=" + singularValues[i]);
//    }
    matU = singularValueDecompositionImpl.getU();
    matV = singularValueDecompositionImpl.getV();
    System.out.println(" svdTolerance = " + singularValueNormalizedTolerance);
    double[] arraySingularValuesNormalized = this.getSingularValuesNormalized();
    numberOfIndenpendentEquations = 0;
    for(int i=0;i<arraySingularValuesNormalized.length;i++){
      if (arraySingularValuesNormalized[i]>singularValueNormalizedTolerance) {
        numberOfIndenpendentEquations = numberOfIndenpendentEquations+1;
        System.out.println(" i = " + i + "  singular value = " + arraySingularValuesNormalized[i]);
      }else{
        System.out.println(" i = " + i + "  singular value = " + arraySingularValuesNormalized[i] +  "   rejected");
      }
    }
    System.out.println(" numberOfIndenpendentEquations = " + numberOfIndenpendentEquations);
    if(numberOfIndenpendentEquations > (numberOfBPMOffsets+numberOfQuads+2*numberOfEnergies)){
       numberOfIndenpendentEquations =  numberOfBPMOffsets+numberOfQuads+2*numberOfEnergies;
    }
  }
  public double[] getSingularValuesNormalized() {
    double maxValue = singularValues[0];
    for(int i=1; i < singularValues.length; i++) {
      if(singularValues[i] > maxValue){
        maxValue = singularValues[i];
      }
    }
    double[] arraySingularValuesNormalized = new double[singularValues.length];
    for(int i=0;i<singularValues.length;i++){
      arraySingularValuesNormalized[i]=singularValues[i]/maxValue;
    }
    return arraySingularValuesNormalized;
  }
  public int getNumberOfIndependentEquations() {
    return this.numberOfIndenpendentEquations;
  }
```

```
  public void solveUsingInverseMatrix() {
    RealMatrix subMatrixV = matV.getSubMatrix(0,matV.getRowDimension()-1, 0, numberOfIndenpendentEquations-1);
    RealMatrix intermediateMatrix = totalMatrix.multiply(subMatrixV);
    QRDecompositionImpl qRDecompositionImpl = new QRDecompositionImpl(intermediateMatrix);
    DecompositionSolver decompositionSolver = qRDecompositionImpl.getSolver();
    double[] intermediateSolutionVector = decompositionSolver.solve(orbitError);

    Array2DRowRealMatrix matrixWithIntermediateSolution = new Array2DRowRealMatrix(intermediateSolutionVector);
    RealMatrix solution = matrixWithIntermediateSolution.preMultiply(subMatrixV);

    xxp_final=new double[numberOfEnergies][2];
    for(int e=0;e<numberOfEnergies;e++){
      for(int i=0;i<2;i++){
        xxp_final[e][i]=solution.getEntry(numberOfBPMOffsets+numberOfQuads+i+2*e,0);
      }
    }
    misQuad = new double[numberOfQuads];
    for(int i=0;i<numberOfQuads;i++) {
      misQuad[i]=solution.getEntry(i, 0);
    }
    misBPM  = new double[numberOfBPMS];
    for(int i=numberOfQuads;i<numberOfBPMOffsets+numberOfQuads;i++) {
      misBPM[i-numberOfQuads]=solution.getEntry(i, 0);
    }
  }

  public void solveUsingPseudoInverseMatrix() {
    RealMatrix SigmaZero = new Array2DRowRealMatrix(numberOfIndenpendentEquations,matU.getColumnDimension());
    for(int i=0;i<numberOfIndenpendentEquations;i++){
      for(int j=0;j<matU.getColumnDimension();j++){
        if(i==j) {
          SigmaZero.setEntry(i, j, 1/singularValues[i]);
        } else {
          SigmaZero.setEntry(i, j, 0);
        }
      }
    }
//    System.out.println("Mat SigmaZero " + SigmaZero.getRowDimension() + "x"+SigmaZero.getColumnDimension());
//    System.out.println("Mat U " + matU.getRowDimension() + "x"+     matU.getColumnDimension());
//    System.out.println("Mat V " + matV.getRowDimension() + "x"+     matV.getColumnDimension());

    RealMatrix sigmaZeroMatUT = SigmaZero.multiply(matU.transpose());
    RealMatrix subMatrixV = matV.getSubMatrix(0,matV.getRowDimension()-1, 0, numberOfIndenpendentEquations-1);
    RealMatrix Pseudoinverse = subMatrixV.multiply(sigmaZeroMatUT);
    RealMatrix ErrorOrbitMatrix = new Array2DRowRealMatrix(orbitError);
    RealMatrix solution = Pseudoinverse.multiply(ErrorOrbitMatrix);

    xxp_final = new double[numberOfEnergies][2];
    for(int e=0;e<numberOfEnergies;e++){
      for(int i=0;i<2;i++){
        xxp_final[e][i]=solution.getEntry(numberOfBPMOffsets+numberOfQuads+i+2*e,0);
      }
    }
    misQuad = new double[numberOfQuads];
    for(int i=0;i<numberOfQuads;i++) {
      misQuad[i]=solution.getEntry(i, 0);
    }
    misBPM  = new double[numberOfBPMS];
    for(int i=numberOfQuads;i<numberOfBPMOffsets+numberOfQuads;i++) {
      misBPM[i-numberOfQuads]=solution.getEntry(i, 0);
    }
  }

  public void solveUsingSimplification() {
    double[] solutionV3 = new double[numberOfBPMS+numberOfQuads+2*numberOfEnergies];
    double[] prod = new double[numberOfIndenpendentEquations];

    for(int k=0;k<numberOfEnergies*numberOfBPMS;k++) {
      for(int i=0;i<numberOfIndenpendentEquations;i++) {
        prod[i]=prod[i]+matU.getEntry(k,i)*orbitError[k];
      }
    }

    for(int j=0;j<numberOfBPMOffsets+numberOfQuads+2*numberOfEnergies;j++) {
      double sum=0;
      for(int i=0;i<numberOfIndenpendentEquations;i++) {
        sum=sum+ (1/singularValues[i])*matV.getEntry(j, i)*prod[i];
      }
      solutionV3[j]=sum;
    }

    xxp_final=new double[numberOfEnergies][2];
    for(int indexEnergy=0;indexEnergy<numberOfEnergies;indexEnergy++){
```

```
    for(int i=0;i<2;i++){
      xxp_final[indexEnergy][i]=solutionV3[numberOfBPMOffsets+numberOfQuads+i+2*indexEnergy];
    }
  }
  misQuad = new double[numberOfQuads];
  for(int i=0;i<numberOfQuads;i++) {
    misQuad[i]=solutionV3[i];
  }
  misBPM = new double[numberOfBPMOffsets];
  for(int i=numberOfQuads;i<numberOfBPMOffsets+numberOfQuads;i++) {
    misBPM[i-numberOfQuads]=solutionV3[i];
  }
}
public double[] getMisQUAD() {
  return misQuad;
}
public double[] getMisBPM() {
  return misBPM;
}
public double[] getMisLaunchforEnergy(int indexEnergy) {
  misLaunch= new double[2];
  misLaunch[0]=xxp_final[indexEnergy-1][0];
  misLaunch[1]=xxp_final[indexEnergy-1][1];
  return misLaunch;
}
public int compareMisalignmentOfQuadsWith(double[] randomOffsetQuads, double MAX_DIFF, boolean showDiffs) {
  int numberOfDiffs = 0;
  for(int iQuad=0; iQuad<numberOfQuads; ++iQuad) {
    double error = misQuad[iQuad] - randomOffsetQuads[iQuad];
    if (Math.abs(error) > MAX_DIFF) {
      if (showDiffs) {
        System.out.println("  misalignment of Quad number " + iQuad + " has been calculated wrong");
        System.out.println("  Calculated : " + misQuad[iQuad] + " Rand: " + randomOffsetQuads[iQuad]);
      }
      ++numberOfDiffs;
    }
  }
  return numberOfDiffs;
}
public int compareMisalignmentOfBPMsWith(double[] randomOffsetBPMs, double MAX_DIFF, boolean showDiffs) {
  int numberOfDiffs = 0;
  for(int iMon=0; iMon<numberOfBPMOffsets; ++iMon) {
    double error = misBPM[iMon] - randomOffsetBPMs[iMon+1];
     // compare 'i' with 'i+1'  because the first and last BPMs are defined to be the reference
    if (Math.abs(error) > MAX_DIFF) {
      if (showDiffs) {
        System.out.println("  misalignment of BPM number " + (iMon+1) + " has been calculated wrong");
        System.out.println("  Calculated : " + misBPM[iMon] + " Rand: " + randomOffsetBPMs[iMon+1]);
      }
      ++numberOfDiffs;
    }
  }
  return numberOfDiffs;
}
public int compareMisalignmentOfLaunchWith(double[][] randomLaunchXXPForEnergy, double MAX_DIFF, boolean showDiffs) {
  int numberOfDiffs = 0;
  for(int iEnergy=0; iEnergy<numberOfEnergies; ++iEnergy) {
    double errorX = xxp_final[iEnergy][0] - randomLaunchXXPForEnergy[iEnergy][0];
    if (Math.abs(errorX) > MAX_DIFF) {
      if (showDiffs) {
        System.out.println("  Launch parameter x of energy " + (iEnergy+1)
        + " has been calculated wrong");
        System.out.println("  Calculated x : " + xxp_final[iEnergy][0] + " Rand: "
        + randomLaunchXXPForEnergy[iEnergy][0]);
      }
      ++numberOfDiffs;
    }
    double errorXP = xxp_final[iEnergy][1] - randomLaunchXXPForEnergy[iEnergy][1];
    if (Math.abs(errorXP) > MAX_DIFF) {
      if (showDiffs) {
        System.out.println("  Launch parameter xp of energy " + (iEnergy+1)
        + " has been calculated wrong");
        System.out.println("  Calculated xp : " + xxp_final[iEnergy][1] + " Rand: "
        + randomLaunchXXPForEnergy[iEnergy][1]);
      }
      ++numberOfDiffs;
    }
  }
  return numberOfDiffs;
```

```
  }
  public double calcDiffRMSofMisalignmentOfLaunchWith(double[][] randomLaunchXXPForEnergy) {
    double[] error= new double[2*numberOfEnergies];
    for(int iEnergy=0; iEnergy<numberOfEnergies; ++iEnergy) {
      error[iEnergy] = xxp_final[iEnergy][0] - randomLaunchXXPForEnergy[iEnergy][0];
    }
    for(int iEnergy=0; iEnergy<numberOfEnergies; ++iEnergy) {
      error[iEnergy+numberOfEnergies] = xxp_final[iEnergy][1] - randomLaunchXXPForEnergy[iEnergy][1];
    }
    double rms = calcRMS(error);
    return rms;
  }
  public double calcDiffRMSofMisalignmentOfQuadsWith(double[] randomOffsetQuads) {
    double[] error= new double[numberOfQuads];
    for(int iQuad=0; iQuad<numberOfQuads; ++iQuad) {
      error[iQuad] = misQuad[iQuad] - randomOffsetQuads[iQuad];
    }
    double rms = calcRMS(error);
    return rms;
  }
  public double calcDiffRMSofMisalignmentOfBPMsWith(double[] randomOffsetBPMs) {
    double[] error= new double[numberOfBPMOffsets];
    for(int iMon=0; iMon<numberOfBPMOffsets; ++iMon) {
      error[iMon] = misBPM[iMon] - randomOffsetBPMs[iMon+1];
    }
    double rms = calcRMS(error);
    return rms;
  }
  double calcRMS(double[] error) {
    return Math.sqrt(StatUtils.variance(error,0.0));
  }
  double calcRMS_selfMade(double[] error) {
    if (null == error) {
      return 0;
    }
    if (error.length < 2) {
      return 0;
    }
    double sumSqr = 0;
    for(int i=0; i<error.length; ++i) {
      sumSqr += error[i] * error[i];
    }
    return Math.sqrt(sumSqr / error.length );
  }
  double calcRMS(double[] array1, double[] array2) {
    if ((null == array1) || (null == array2)) {
      return 0;
    }
    if (array1.length != array2.length) {
      return 0;
    }
    if (array1.length < 2) {
      return 0;
    }
    double sumSqr = 0;
    for(int i=0; i<array1.length; ++i) {
      double diff = array1[i] - array2[i];
      sumSqr += diff * diff;
    }
    return Math.sqrt(sumSqr / array1.length );
  }
}
```

# References


[1] M. Borland. *ELEGANT: A flexible SDDS-compliant code for accelerator simulation.* Aug 2000.

[2] E.D Courant and H.S Snyder. Theory of the alternating-gradient synchrotron. *Annals of Physics*, **3**(1):1 – 48, 1958.

[3] Winfried Decking, Torsten Limberg, and Hyunchang Jin. Beam-based alignment in the european xfel sase1; tesla-fel 2013-03. Technical report, Deutches Elektronen-Synchrotron (DESY), 2013.

[4] Helmut Wiedemann. *Particle accelerator physics.* Springer, New York, 2007.