Faster Optimal Design Calculations for Practical Applications

Eric Strömberg

Pharmacometrics Research Group
Department of Pharmaceutical Biosciences
Uppsala University
Sweden
Abstract

**Faster Optimal Design Calculations for Practical Applications**

_Eric Strömberg_

PopED is a software developed by the Pharmacometrics Research Group at the Department of Pharmaceutical Biosciences, Uppsala University written mainly in MATLAB. It uses pharmacometric population models to describe the pharmacokinetics and pharmacodynamics of a drug and then estimates an optimal design of a trial for that drug. With optimization calculations in average taking a very long time, it was desirable to increase the calculation speed of the software by parallelizing the serial calculation script. The goal of this project was to investigate different methods of parallelization and implement the method which seemed the best for the circumstances. The parallelization was implemented in C/C++ by using Open MPI and tested on the UPPMAX Kalkyl High-Performance Computation Cluster. Some alterations were made in the original MATLAB script to adapt PopED to the new parallel code. The methods which where parallelized included the Random Search and the Line Search algorithms. The testing showed a significant performance increase, with effectiveness per active core ranging from 55% to 89% depending on model and number of evaluated designs.

Sammanfattning

**Snabbare Optimal Design beräkningar för Praktiska Applikationer**

_Eric Strömberg_

PopED är en mjukvara utvecklad av Farnakometriska Forskningsgruppen vid institutionen för Farmaceutisk Biovetenskap, Uppsala Universitet och är skriven mestadels i MATLAB. Den används till att estimera en optimal design av kliniska prövningar för ett läkemedel genom att använda sig av farmakometriska populationsmodeller. Med beräkningsmässigt tunga modeller och stora populationer kan denna optimering ta mycket lång tid då PopED exekveras seriellt. Därför var det önskvärt att parallellisera de beräkningsalgoritmier som PopED använder.

Målet med detta projekt var att undersöka olika parallelleringsmetoder och implementera den metod som tycktes passa bäst för syftet. Parallellerisingen implementerades huvudsakligen i C/C++ genom att använda Open MPI, dock så gjordes det en del ändringar i PopEDs MATLAB skript för att anpassa detta till de parallella algoritmier. De metoder som hade huvudsaklig fokus var de linjära och randomiserade sökalgoritmier. Tester utfördes på UPPMAX Kalkyl beräkningscluster och påvisade en signifikant ökning i hastighet hos optimeringarna, där effektiviteten per arbetande CPU-kärna varierade mellan 55% och 89%, beroende på modell och antalet evaluerade designer.
Contents

1. Introduction ........................................................................................................................................ 4
   1.1 Pharmacokinetics ............................................................................................................................ 4
   1.2 Pharmacodynamics .......................................................................................................................... 5
   1.3 Pharmacometrics and pharmacometric models .................................................................................. 5
   1.4 Clinical Trials .................................................................................................................................. 6
   1.5 Population Experimental Design(PopED) ....................................................................................... 8

2. Theory .................................................................................................................................................... 9
   2.1 Population model and Fisher Information Matrix ........................................................................... 9
   2.2 Optimal Design ............................................................................................................................... 12
   2.3 Parallel Computing .......................................................................................................................... 14

3. Methods of Parallelization of PopED ................................................................................................. 17
   3.1 MATLAB™ Parallel Computing Toolkit® (PCT) ............................................................................. 17
   3.2 Open MPI ......................................................................................................................................... 22
   3.3 Other Considered MPI solutions .................................................................................................... 24

4. Implementation of parallelization of PopED with Open MPI ............................................................. 25
   4.1 Changes in MATLAB functions ...................................................................................................... 26
   4.2 The Job Manager/Worker ............................................................................................................... 29
      4.2.1 Job manager ............................................................................................................................. 32
      4.2.2 Worker ..................................................................................................................................... 34
      4.2.3 Additional Dependencies ....................................................................................................... 35

5. Testing .................................................................................................................................................. 37

6. Results .................................................................................................................................................. 39

7. Conclusion ........................................................................................................................................... 41

8. Discussion ............................................................................................................................................ 41

9. Acknowledgements ............................................................................................................................ 44

10. References .......................................................................................................................................... 44
1. Introduction

With the costs of drug development increasing dramatically, it has become more common for not only the pharmaceutical companies, but the regulatory authorities as well, to look towards alternative methods for reducing costs. One of the main culprits in this rapid increase of development costs has been the increasing requirements for verified drug efficacy, safety and adverse effect documentation produced in the expensive clinical trials part of the development process. Thus it has become desirable to reduce the cost of performing clinical trials, without reducing the statistical power and accuracy of the results, which would put the consumers in greater risk.

One method to achieve the above, which has become highly desired within the pharmaceutical industry, is to optimize the design of the clinical trials in terms of variables such as sample time and group size. To simplify this procedure a software PopED has been developed by the Pharmacometrics Research Group at the department of Pharmaceutical Biosciences at Uppsala University. At the current version of PopED the optimization algorithm still operates in serial and thus does not utilize the full potential of the multicore computers or cluster systems of today. This leads to long run-times for computational intensive optimizations and large number of designs to be evaluated, which is common. One solution to increase the speed is to parallelize PopED so it uses all the available cores or CPUs and distributes the workload across these resources.

Below follows a short introduction to the area of quantitative pharmacology.

1.1 Pharmacokinetics

Pharmacokinetics is a mathematical approach to describe how the body affects an administered drug by combining parameters that explains the drugs current state and concentrations at a given time t. The most common parameters which are discussed can be found in table 1.

Table 1: Common parameters in pharmacokinetic calculations.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cp</td>
<td>Plasma Concentration</td>
<td>g/L</td>
</tr>
<tr>
<td>CL</td>
<td>Clearance</td>
<td>L/h</td>
</tr>
<tr>
<td>V</td>
<td>Volume of distribution</td>
<td>L</td>
</tr>
<tr>
<td>Ka</td>
<td>Absorption rate constant</td>
<td>h⁻¹</td>
</tr>
<tr>
<td>T1/2</td>
<td>Half-life</td>
<td>h</td>
</tr>
<tr>
<td>τ</td>
<td>Dosing interval</td>
<td>h</td>
</tr>
<tr>
<td>F</td>
<td>Bioavailability</td>
<td>n.u. (%)</td>
</tr>
</tbody>
</table>
Parameters can be approximated empirically with a non-model based approach by measuring plasma concentration of the administered drug and plotting the concentrations against time. The area under the curve can provide the basis for calculating the parameters, in turn, the AUC is often approximated with the trapezoidal rule or more advanced numerical methods. However, it is also useful to estimate the parameters by using a model-based approach where the parameters have their values estimated to best fit the data.

1.2 Pharmacodynamics
Pharmacodynamics is often interlaced with pharmacokinetics and is used to describe how the achieved concentration of drug in the circulatory system affect the body by applying effect models which correlate e.g. drug-concentration in plasma with the physiological response or a biomarker level. A biomarker is a quantifiable entity which has its magnitude changed as a result of stimulus from e.g. a drug or disease progression and they can be related to a physiological effect. Biomarkers are commonly used to monitor the effect instead of the physiological response since they might be easier to quantify. Common parameters which are used to describe a drug efficacy, or ability to induce a physiological effect include EC50 or IC50, which are used to describe the concentration which results in a 50% of maximum effect (EC50) or 50% of maximum inhibition (IC50). Lower EC50/IC50 corresponds to a more potent drug with a higher affinity to form the drug-receptor complex which leads to an effect.

1.3 Pharmacometrics and pharmacometric models
A relatively new discipline in the area of quantitative pharmacology is pharmacometrics. This is an area where expertise from several fields e.g. pharmacology, pharmacokinetics/pharmacodynamics, statistics and engineering are combined to better understand the data generated from a trial or study.

Here it is common to not only consider the raw data and the variance within one individual, but also to incorporate the variance between individuals, i.e. a population. This makes it possible to visualize the entire population in terms of a confidence interval in which all the individuals, at a specific statistical probability, should be found. By describing the data in such manner it might better describe how certain individuals can achieve e.g. higher degree of adverse effects than others.
An interesting approach to extend the informative region of the generated data is to use a model based approach to the data analysis. Pharmacometric models apply the mathematical correlations discussed in pharmacokinetics/pharmacodynamics to create simplified models of the drugs passage in the body. For instance, the drugs distribution in different regions in the body can be described by simple diffusion with a specified rate between compartments and then the concentration can be measured in the compartment of interest.

In Figure 1 a simple two-compartment model is represented by drawing boxes which symbolize the compartments; Central, usually systemic circulation, and periphery, e.g. fatty tissue. The rate of diffusion between the compartments are given by the $k$-value and the rate of absorption is dependent on $k_{abs}$.

By fitting a Pharmacometric model to existing data the model can extrapolate the data into informative regions that earlier where not available, and thus increase the information generated by the experiment. Although you must always consider a famous quote uttered by George Edward Pelham Box, a Professor Emeritus in statistics at the University of Wisconsin, which has effectively become one of the most important rules and topics of discussion within pharmacometrics:

"Essentially all models are wrong, but some are useful"

This quote captures the goal of every pharmacometric model, which is to describe something as accurately as possible, including the error of the models description.

1.4 Clinical Trials
Since the first drugs emerged people have been craving evidence of the drugs effect and safety, in the older days the experience of a practitioner such as a doctor or the local healer was enough. But with the drugs getting a more central and scientific role in the society more substantial evidence has been required. This has resulted in the practical clinical trial, which standards where set back in the 1960’s and has not changed much until recently. The clinical trials has for decades had simple endpoints, such as an observed effect or change in a disease progression. And this with keeping adverse effects in mind, something that got radically more
attention since the Thalidomide incident from the early 1960’s when its teratogenic adverse effect was noticed. Collecting data from the clinical trials had long before this been of the highest scientific interest, but Thalidomide induced a higher regulatory demand of the developed drugs, and thus introduced more extensive trials with bigger populations to decrease the probability of missed adverse effects. The clinical trials which most drugs have to pass are divided into four phases as described below [1]:

**Phase I:** First administration of the candidate drug in man, thus requiring a highly controlled environment where the healthy volunteers usually are constantly monitored for several half-lives of the drug. It is here that data for the first human pharmacokinetic/pharmacodynamic parameters is generated. The primary goal is to assess tolerability of the drug and to determine the appropriate therapeutic dose-range to be tested in the next phases.

There are different types of trials within phase I where either a single dose is administered (SAD: Single Ascending Dose) or several doses are administered as a dose regimen corresponding to similar peak concentrations (MAD: Multiple Ascending Dose). The patients are monitored for adverse effects whilst collecting data. SAD is commonly used to determine the MTD (Maximum Tolerated Dose), e.g. where the adverse effects become intolerable. MAD is conducted to give a better picture of the PK/PD aspect of the drug which helps determining the dose regimen that might be useful in practice.

It is also common to control food effect on the PK/PD properties of the drug in phase I. Here a crossover study is useful, where subjects are divided into several groups. One, or more, of these groups will receive a standardized meal, while others will be left unfed. After the drug has been administered, the groups will be monitored and the data will be compared to derive an eventual effect of food intake.

The test-population for phase I trials is most commonly smaller groups of healthy volunteers, but due to ethical restrictions there are some cases, e.g. Cancer, where patients are used in phase I trials. The volunteers will usually receive a small monetary compensation for participating in the trials. The size of the compensation depends on how invasive the procedure is.

**Phase II:** When the tolerability and PK/PD properties of the candidate drug is established from phase I the drug move on to the next phase, which include another test-population where the individuals are the target population for that drug. Here the proof-of-concept (POC) must
be achieved for the development to continue. This is to prove that the drug actually works as intended in the body and that there are no intolerable adverse effects of the drug within the target population. It is in phase II that the recommended dose regimen for the drug is established, which will be used for the next large scale testing phase. Phase II can be divided into phase IIa and phase IIb where the dose regimen is developed in the first phase and the efficacy of that regimen is tested in phase IIb.

Phase III: The large scale phase where the test-population is the largest and the time of the trial is the longest. It is in this phase that the drug is tested on a widespread population on different locations and it is here that the costs of the drug-development escalate. The main reason for the large and variable population is to achieve data as representative as possible in term of describing the global population. Ideally the data would contain all the variability that can be found in the entire human population so that every individual is represented and thus no occasional intolerable adverse effect or drug inefficiency is missed. Groups of patients will receive drug and then be compared to other groups and groups that received placebo to assess the average drug efficacy. Although in some cases with chronically ill patients it is unethical to administer placebo so the effect of the suggested dose regimen of the candidate drug is here compared to the effect of either a lower effective candidate drug concentration or more commonly, another effective drug. If that is the case the drug can be deemed non-inferior or superior depending on the design and desired outcome.

After successful phase III trials which meet up with the medical products agencies standards the application for the approval of the new drug is formed. FDA (Food and Drug Administration in USA) and EMA (European Medicines Agency) both requires two successful large scale phase III trials as a minimum.

Phase IV: After the drug has been released it is usually monitored post market by collecting any reports of adverse effects and evaluating submitted case reports from clinics. The effects and safety of the drug is still considered, but it is also common to look for any new indications where the drug might be effective. Exactly where the phases IV ends and continue into regular post marketing surveillance of the product is not definite.

1.5 Population Experimental Design(PopED)
One tool for optimization of trial designs by using population modeling is PopED which was first introduced in 2004 [2]. It is software developed by the Pharmacometrics research group at the department of Pharmaceutical Biosciences, Uppsala University. PopED consist of a
GUI (Graphical User Interface) written in C# which is interlaced with MATLAB-script through XML. The optimization, differential equation calculations and plot-managements are all handled within the MATLAB engine.

The user inputs the pharmacometric model as a MATLAB-script in the PopED GUI and edits optimization settings. The software will then generate the required model in MATLAB script function (.m) files and the optimization settings in an XML-format. The MATLAB engine will next import the settings in the XML-file and translate into a .m MATLAB function file. The optimization or evaluation of the design(s) will then be performed within MATLAB. The script has several different algorithms for optimization, including D-optimal and ED optimal design (see section 2.2). PopED will calculate the FIM (Fischer Information Matrix) for each design evaluated, which is the majority of the computational load during the optimization. Upon completion of the calculation MATLAB will plot results and, if required, save the results in a XML-file for later use.

Since PopED uses the MATLAB engine it is possible to manage both simple models and advanced differential equations with e.g. the built-in ODE-solvers that MATLAB support. When the models becomes more advanced, the time to complete the optimization increases. It is not uncommon that models can take weeks or months to optimize when PopED is running in serial, which does not take advantage of the full computational power of e.g. the HPC-cluster system at the department. Thus it is highly desirable to parallelize the optimization script of PopED to increase the speed.

2. Theory

2.1 Population model and Fisher Information Matrix

The mathematical definition according to Nyberg et al. [3] of a population model can be described accordingly for resulting observations of an experiment with individuals i:
Where $\tilde{y}_i$ is the vector of responses from the $i$th individual generated by a function $f_j(...)$ which describes how the response $j$ changes with the experimental variables $\tilde{t}_{ij}$ and the individual parameters $\tilde{\beta}_i$. The residual error model of response $j$ is given by the function $h_j(...)$, where $\tilde{e}_{ij}$ is the vector of residual error terms which is normally distributed with zero mean and a variance $\Sigma$ such as:

$$\tilde{e}_{ij} \sim N(0, \Sigma), \quad \Sigma = \begin{pmatrix} \sigma_1^2 & 0 \\ 0 & \ddots & 0 \\ 0 & \ddots & \sigma_n^2 \end{pmatrix}$$

(2)

Where $n$ is the total number of residual error parameters.

The individual parameters for the $i$th individual can be described as:

$$\tilde{\beta}_i = g(\tilde{\theta}, \tilde{\eta}_i, \tilde{a}_i)$$

(3)

The individual parameters can be said to be dependent on a function $g$ which describes the variability and transformation of said parameters where $\tilde{\theta}$ is a vector with the typical values of the model, i.e. fixed effect parameters. Furthermore, the function output is dependent on the known covariates, represented by a vector $\tilde{a}_i$ and the between subject variability (BSV) $\tilde{\eta}_i$, which is considered normally distributed with zero mean and variance $\Omega$ accordingly:

$$\tilde{\eta}_i \sim N(0, \Omega), \quad \Omega = \begin{pmatrix} \omega_1^2 & 0 \\ 0 & \ddots & 0 \\ 0 & \ddots & \omega_k^2 \end{pmatrix}$$

(4)

Where $k$ is the total number of BSV parameters.

As mentioned above it is desired to estimate the population parameters from the experimental data generated. Thus it is assumed that all $\tilde{\eta}_i$ are independent of each other and of the residual error $\tilde{e}_i$.

According to the Cramer-Rao inequality [4], the lower bound of the variance-covariance matrix of the parameters in the model can be derived by the inverse of the Fisher Information Matrix:
FIM = \(-E\left[ \frac{\partial^2 L(\hat{\theta})}{\partial \hat{\theta} \partial \hat{\theta}^T} \right] \), \quad \hat{\theta} = [\hat{\theta}, \hat{\omega}, \hat{\sigma}] 

Where FIM is the expectation value over data of the second derivate of the negative joint log likelihood function $L$ with respect to the estimated parameters given by $\hat{\theta}$. This has been described previously by Forracchia et al. [2] and the method which is used to compute FIM is based on that approach, although here slightly modified.

One way of assuring that the random effects are normally distributed is to linearize $f(\ldots)$ and $h(\ldots)$ with respect to $\eta$ and $\varepsilon$. This can be done with a first order (FO) approximation, which linearizes around the mean values of $\eta$ and $\varepsilon$, i.e. $\bar{\eta}_i = 0$ and $\bar{\varepsilon}_i = 0$, which with Equation 1 and Equation 3 gives:

$$\bar{y}_i = f(\tilde{t}_i, g(\bar{\theta}, \bar{\eta}_i, \bar{a}_i)) + h(\tilde{t}_i, g(\bar{\theta}, \bar{\eta}_i, \bar{a}_i), \bar{\varepsilon}_i) \\
\approx f(\tilde{t}_i, g(\bar{\theta}, 0, \bar{a}_i)) + L_i(\tilde{t}_i, g(\cdot)) \cdot \bar{\eta}_i + H_i(\tilde{t}_i, g(\cdot), \bar{\varepsilon}_i) \cdot \bar{\varepsilon}_i \\
= f(\tilde{t}_i, g(\bar{\theta}, 0, \bar{a}_i))$$

(6)

$$L_i(\tilde{t}_i, g(\bar{\theta}, \bar{\eta}_i, \bar{a}_i)) \equiv \frac{\partial f(\tilde{t}_i, g(\bar{\theta}, \bar{\eta}_i, \bar{a}_i)^T)}{\partial \eta_i} \bigg|_{\bar{\eta}_i = 0}$$

(7)

$$H_i(\tilde{t}_i, g(\bar{\theta}, \bar{\eta}_i, \bar{a}_i), \bar{\varepsilon}_i) \equiv \frac{\partial h(\tilde{t}_i, g(\bar{\theta}, \bar{\eta}_i, \bar{a}_i)^T, \bar{\varepsilon}_i)}{\partial \varepsilon_i} \bigg|_{\bar{\varepsilon}_i = 0}$$

(8)

Unlike the model presented by Forracchia et al. [2] and Retout et al. [5] this model allow error structures other than the proportional and additive. The expected mean and the corresponding variance of the responses from the linearized model can be described as shown in Equation 9 and Equation 10 respectively:

$$E(\bar{y}_i) \approx f(\tilde{t}_i, g(\bar{\theta}, 0, \bar{a}_i))$$

(9)

$$\text{Var}(\bar{y}_i) \approx L \cdot \Omega \cdot L^T + \text{diag}(H \cdot \Sigma \cdot H^T)$$

(10)

Finally the individual FIM with respect to the parameters $\bar{\Theta}$ and design variables $\bar{x}_i$ can be defined according to Equation 11.

$$\text{FIM}_l(\bar{\Theta}, \bar{x}_i) = \begin{bmatrix} M_{11} & 0 \\ M_{21} & M_{31} \end{bmatrix} \begin{bmatrix} V_i^{-1} & 0 \\ 0 & M_i^{-1} \end{bmatrix} \begin{bmatrix} M_{11} & 0 \\ M_{21} & M_{31} \end{bmatrix}^T$$

(11)
\[ M_{1i} = \frac{\partial f(t_i, g(\theta, b, \tilde{a}_i))}{\partial \theta} \]  

(12)

\[ M_{2i} = \frac{\partial \text{vec}(\text{Var}(\tilde{y}_i))}{\partial \theta} \]  

(13)

\[ M_{3i} = \frac{\partial \text{vec}(\text{Var}(\tilde{y}_i))}{\partial \tilde{a}^2} \]  

(14)

\[ M_{4i} = 2\text{Var}(\tilde{y}_i) \otimes \text{Var}(\tilde{y}_i) \]  

(15)

Where \( x_i = [\tilde{t}_i, \tilde{a}_i] \).

Furthermore, the population FIM can be considered the sum of all individual Fisher information matrices:

\[ \text{FIM} \left( \bar{\theta}, \bar{x} \right) = \sum_{i=1}^{m} \text{FIM}_i(\bar{\theta}, \bar{x}_i) = \sum_{i=1}^{N_g} g_i \cdot \text{FIM}_i(\bar{\theta}, \bar{x}_i) \]  

(16)

\[ m = \sum_{i=1}^{N_g} g_i \]  

(17)

Where \( m \) is the total number of individuals, \( g_i \) is the number of individuals in each group and \( N_g \) is the total number of groups.

### 2.2 Optimal Design

No experiment is perfectly executed; it always possible to tweak and improve a study to improve on how informative the results of the experiment is. This is a well known fact within the field of experimental science. One way to achieve this improvement is to think ahead and by using knowledge from previous, similar, experiments tweak the coming experimental design before it is performed. This is what optimal design is all about: To plan a study optimally to increase the information gained.

Within the clinical practice there is a great variability in the data, not only dependent on the subjects themselves, but also depending on how the practitioners have performed when collecting the data. It was in early clinical trials only interesting to monitor the condition improvement and occurrence of adverse events, which did not require much planning regarding how the sampling should be done.

It has always been in everyone’s greatest interest to not only have total control, but to gain as much information as possible whilst minimizing the costs. By applying an optimal design to a trial the information gained might increase significantly. And the golden egg for most
pharmaceutical companies is the idea that an optimized trial with a smaller test-population might be even more informative than a non-optimal trial with a greater test-population, which could reduce costs. This is one of the main reasons why pharmacometricians with an interest towards optimizing the clinical trial design has become increasingly attractive in the pharmaceutical industry.

Consider the plot of drug-concentration in central compartment vs. time seen in Figure 3. If the PK-profile was accurate for a phase I trial where the PK-parameters for the drug in human is estimated by measuring the concentrations of drug in plasma at certain times, then the result will vary depending on when the samples were taken. The goal is to capture as much as possible of the characteristics of the PK-profile of the drug. For example; if the sampling times are at relatively uninformative points in the curve it might result in a faulty model. For instance, if four sample-times are at the inclination of the curve, before the maximum, and the last sample time is at the very end of the time scale, the data will not represent the PK-characteristics of the drug in a good way. This might lead to inaccurate estimates for the vital parameters, which then applied to the design of a clinical trial might result in e.g. unacceptable drug concentrations in the circulatory systems or adverse events which could have been avoided with a better design.

There are several different criteria of optimal design. The two main methods, used by PopED, are the D-optimal and ED-optimal designs, with the main focus on D-optimal in this project.

The D-optimal design aims to minimize the expected variance and covariance by maximizing the determinant of the Fisher information matrix described above. As mentioned previously; the information matrix will according to the Cramer-Rao provide the asymptotically lower bound value of the Variance-Covariance matrix. Since the determinant of the FIM is used, the optimizations will consider all the elements of the FIM [3]. The efficiency, $D_{eff}$, of one optimized design, $\tilde{x}_1$, in comparison to another design, $\tilde{x}_2$, can be determined according to Atkinson-Donovlev [6]:

![Figure 3: Concentration profile generated by a two-compartment model with first order absorption. Created with Berkley-Madonna®](image)
\[ D_{\text{eff}} = \left( \frac{\text{FIM}(\hat{\theta}, \hat{x}_1)}{\text{FIM}(\hat{\theta}, \hat{x}_2)} \right)^{1/p} \]  

(18)

Where \( p \) is the total number of parameters in the design and \( \hat{\theta} \) are the model parameters as described previously.

Essentially optimal design software, such as PopED calculating a D-optimal design, is actually doing an automated global search for the design which generates the maximum value of the determinant of FIM. Two of the search algorithms used in PopED to achieve a global optimum are the Line Search (LS) and the Random Search (RS).

In the LS the possible design space is divided into a grid of equally distributed values in each design dimension. The design in the grid with the highest FIM (according to the criteria, e.g. D-optimal) is stored and recorded as the most informative design.

Random search executes in a similar manner, although for RS the value of the design parameters are randomized within the defined range. This can lead to a less accurate result if there are few random steps made. Random search is often combined with LS by having the search algorithms repeated until the results of LS and RS converge within a predefined criterion. Both RS and LS converge to a global optimum when the number of iterations (\( n \)) in the search algorithm goes towards infinity. However, a large \( n \), is in general not possible due to computer runtime and hence a local search method (e.g. gradient method) is needed. In this work we focus on the global search methods LS and RS.

2.3 Parallel Computing

With the increase of multi-core CPU:s that has become available it is not only in academia desirable to have computational heavy applications run in parallel. With the introduction of Intel Corporation’s first multi-core processors to the general public, software developers has been looking at efficient ways of utilizing the increased power. Hardware manufacturers have since then produced increasingly advanced multi-core solutions faster than software developers have been able to adapt their applications to the new parallel environment by parallelizing their serial software.

Multithreading and parallelized software are two common statements regarding software, with two different meanings. Multithreading essentially means that software can use multiple threads over a CPU which has been commonly used to handle the different strings of data which has to pass over the CPU to other parts of the computer. Parallelized software on the
other hand can distribute its load over those different threads and then distribute the threads over different cores to balance the computational load more or less efficiently, as shown in figure 4.

![Figure 4: Schematics of a process divided into n computational threads Ti which is handled with serial and parallel computation. Here, each CPU-core under the parallel software have one worker. Job manager (blue background) collect and send work to workers.](image)

To achieve fully parallelized software a message passing interface (MPI) is needed. This computational sublanguage is used to communicate between the running process and the distributed working threads. Basically telling the different working threads what they are supposed to do and also receiving the result from the workers.

It is common to describe parallelized software as an applications workload being distributed across a number of “workers” with one process acting as a “job manager” which sends work and receives results with the MPI. Usually one worker per CPU-core is chosen with preferably the job manager on an independent core.

The theoretical ideal gain in speed would simply be a factor equal to the number of CPU-cores over which the application is parallelized. In reality the MPI communications somewhat bottlenecks the ideal since it is dependent on how quickly packets can be sent/received between workers and the process manager. Often, additional cost, as e.g. work initializing time, will further decrease efficiency.
Two main hardware-dependent architectures for parallel computing are often taken in consideration [7]:

**Distributed Memory Architecture:** Processes get distributed over a grid of separate physical memory for each node that exists within the calculation unit, e.g. a HPC- Cluster. All running processes have their own address space within the memory and thus cannot access other running processes’ data without message passing interface. Thus process synchronization and management is all up to the programmer to sort out in an efficient manner. Each processor can quickly access its own memory by default, which makes the total cluster memory completely scalable with number of computational nodes and the system easily upgraded with standard processors and networking.

**Shared Memory Architecture:** All processors can access the memory as a shared global memory, and thus changes made to data by one process becomes visible to all other processes. This makes sharing of data both fast and user friendly, but since the memory is not completely scalable with the number of added nodes, the system is not as easily upgraded as a distributed memory architecture system. There are two different types of shared-memory architecture denoted UMA (Uniform Memory Access) and NUMA (Non-Uniform Memory access) in which the CPU’s either have one global memory or local bus-interconnected memory respectively.

![Schematics of different memory architectures for parallel computing](image)

*Figure 5: Schematics of different memory architectures for parallel computing: Distributed Memory Architecture at the top and Shared memory architecture with UMA(Uniform Memory Access) and NUMA (Non-Uniform Memory Access) to the down left and down right respectively.*
3. Methods of Parallelization of PopED

During the first weeks of the project several potential ways of parallelizing the optimization script of PopED were investigated. These varied from finished extensions of MATLAB, easily implemented within the existing MATLAB code to achieve parallel computation of the FIM-calculation, to advanced open-source MPI-libraries which would require an in-depth approach to parallelize PopED. The most interesting approaches and their pros and cons are presented below, including the method which was chosen.

3.1 MATLAB™ Parallel Computing Toolkit® (PCT)

The first method which was investigated acts as an MPI-addon to the existing MATLAB API and is developed by Mathworks. This is an, for native MATLAB applications, rather effective and simple way of parallelizing existing calculations since it requires few changes to existing code. One of the easiest examples of this is when a simple linear for-loop is to be parallelized using the command \texttt{parfor} as shown in Example 1.

\begin{example}
\textbf{Example 1: Parallel for-loop in MATLAB using parfor.}
\begin{verbatim}
function []= parForTest()

M = 200;
N = 300;

% Vectors that store the maximum eigenvalues
a = [];
%serial
b = [];
%parallel

% serial (regular) for-loop
for i = 1:N
    a = [a max(eig(rand(M)))];
end

% parallel for-loop
matlabpool open 2
parfor j = 1:N
    b = [b max(eig(rand(M)))];
end
matlabpool close
\end{verbatim}
\end{example}
In this example a for-loop elongates and fills a vector of length N with the maximum eigenvalues of N randomized MxM matrices. The values of M and N has been set to 200 and 300 respectively, but it can easily be changed to either increase the number of iterations or increase the computational load of each iteration of the for-loop. The example also contains a segment with a parallel parfor-loop which does similar computations as the for-loop. The only required changes to the code in this case is to simply open a matlabpool, call parfor instead of for and finally close the matlabpool after the parallel work is done.

The matlabpool is the PCT’s command to initialize the workers necessary to execute a parallel computation. In this case a pool of two workers is opened, one for each CPU-core if one has a dual-core CPU, the number of workers can easily be changed by altering the number following matlabpool open. The pool will then have to be closed either in order to save RAM on the computer if the parallel work is done, or if another pool is to be opened. MATLAB PCT includes a job manager which automatically does a load balance over the available workers as the default setting. If required the job manager can be made interactive, and thus give the user control of each workers load and data. The load balance can as well be done in different ways, as it has been done here the number of iterations of the for-loop is simply split between the two workers, giving them 150 iterations each, which then gets collected by the job manager in the background and saved as the final b-vector. Another option would have been to create a distributed array, which basically would have created independent parts of the MxM array on different workers. In this example the distributed array would correspond to having a matrix of dimensions (0.5xM)xM , 100*200 at the current setting of M, on each worker. This is a good example of that the MATLAB Parallel Computing Toolkit actually works with distributed memory architecture. [8]

The eventual increase of speed is always the primary concern, and the above example did not really capture the differences in a clear manner. Thus the function was altered to store the iteration time in vectors called t1 and t2 for the serial and parallel loop respectively. The results was finally plotted to illustrate the results, which can be seen in Figure 6.

The two graphs both show each time-point where a calculation has been finished. In this specific example it is clear that the parallelized version of the for-loop performs at almost twice the speed as the serial loop since the length of dots representing finished calculations by the parallel loop (green) ends at nearly half the length of the dots representing the serial loop.
The bottom plot is simply a magnified region of the above plot, which makes it easy to see that for each iteration finished by the serial loop, approximately two iterations of the parallel loop finishes.

![Graph](image)

*Figur 6: Plots made in MATLAB to illustrate at which times the for and parfor loops finishes their iterations. The plot below shows a magnified region of the approximate area marked with a red box above.*

The PCT also allows the user to implement so-called *SPMD*-blocks (Single Process Multiple Data), which basically tells the engine that within this block the code can be run simultaneously on several workers to balance the workload. This is one of the standard distributed memory architecture techniques and a parallel example which require message passing. For this to be used efficiently in MATLAB the user can assign which worker(s) that is supposed to handle certain parts of the code within the block. The workers are initialized with the `matlabpool` command as in the previous example. It is also possible to have real-time control of the processes through a graphical interface which resembles a set of MATLAB command windows, one for each worker. This can be useful for testing whether a MATLAB function actually can run efficiently in parallel or not.
Since the optimization script of PopED is highly parallelizable on the FIM calculation level which consists of a series of for-loops, interchangeable with the parfor-loop, the first obvious way of parallelizing PopED would be to use the parfor-loop. Thus it was investigated how this speed increase would depend on the number of designs to be evaluated by PopED, e.g. how the relative speed-up depends on the number of iterations and calculative load of each iteration. To illustrate this dependency, the previous example parForTest.m was altered to do a sweep with different sizes and different numbers of iterations and then plot the differences. The full code of this function parForSweep() can be found in Appendix A.

![Graph](image)

**Figure 7**: Time-to-completion plotted with increasing number of iterations (increasing N). Graph contains both parallel examples on dual-core and quad-core systems. The relative increase of speed and the original non-parallelized result is plotted as well.

As seen in Figure 7 and Figure 8 the relative increase of computational speed does not increase linearly as the number of active cores increase. In this specific case it should be mentioned that the difference in result between the dual-core and quad-core is not exactly fair since they were both made on the same quad-core CPU. This would lead to the quad-core result to be less than it actually is since the CPU core-load would be higher due to background applications. In the case of the single-core and dual-core applications the CPU would
automatically direct the workload to the least active cores for increased performance. For a more accurate result these same plots would have to be made on a at least penta-core, preferably hexa-core, system.

![Graph showing elapsed time for different loop configurations](image)

*Figure 8: Elapsed time until completion of all loops (5050 iterations).*

Nevertheless those results would probably neither show an exactly linear increase of speed since the message passing between cores still requires some time. Although it should also be considered that the parallelization is the most effective for computational demanding calculations, since those take more time relative to the message passing time than the lighter calculations. Thus it has also been tested to have few, but computational demanding iterations, which showed similar results to Figure 7 and Figure 8. The difference between dual-core and quad-core increased somewhat, but the results were still affected by the same degeneration caused by the available CPU and running sub-processes as the previous tests. To eliminate this factor the final runs should be performed on a dedicated HPC-cluster.

So far, all testing, had been done on a regular desktop. The main reason for this is that for HPC-clusters the MATLAB Parallel Computing Toolkit limits the maximum number of workers to four. The only available workaround for this is to purchase and install the MATLAB® Distributed Computation Server™ (DCS). This provides the user with a default jobmanager and lightweight MATLAB ghost-clients which acts as workers, similar to the MATLAB Compiler Runtime. The MATLAB DCS is sold in multiples of eight workers. Since this requirement for efficiently running tests on the resident cluster was unavailable at
the time of testing it had to be skipped. The results were considered sufficient by the supervisors and the advantages and disadvantages of MATLAB Parallel Computing Toolkit is presented in Table 2.

Table 2: Advantages and Disadvantages with using MATLAB® Parallel Computing Toolkit™

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relatively Easy Implementation compared to other MPI-solutions</td>
<td>Less control of level of parallelization than other MPI-Solutions</td>
</tr>
<tr>
<td>Performance would scale well in theory</td>
<td>Performance might be less than more thorough parallelization from e.g. Open MPI</td>
</tr>
<tr>
<td>Great amount of available documentation</td>
<td>Non-open Source solution</td>
</tr>
<tr>
<td>Available multi-lingual support from both Mathworks and the MATLAB® community</td>
<td>Requires MATLAB® Distributed Computation Toolkit™</td>
</tr>
</tbody>
</table>

3.2 Open MPI

Opposed to the copyrighted and rather restricted MATLAB Parallel Computing Toolkit, several open source alternatives exist to use interlaced with MATLAB’s C/C++ compatibility. One of these options which was considered is Open MPI, not to be confused with OpenMP which is developed by another MPI consortium. Open MPI, often abbreviated OMPI, is an open source library for several computational languages such as C/C++ and FORTRAN which is developed by a consortium of academic and retail MPI professionals with the goal of combining the knowledge into a free and universal MPI solution [9]. The current version is based on the MPI-2 standard and contains all the native functionality from MPI-2 [10] together with the extended libraries developed by the Open MPI consortium to form the complete solution. This is available for most platforms and requires the developer to not only build and compile the Open MPI libraries and client, but also to implement the message passing and parallel communication in the application to be parallelized. Thus it is a bit more complicated than with MATLAB PCT, to get the required client for parallel computing used by Open MPI, spmd.exe, installed on the computer system used for testing. However, once installed there is quite a lot of documentation and examples to ease the understanding of how to use Open MPI and to implement parallel software, e.g. PopED.

One of the provided examples by the Open MPI developers is ring_c.c. This example basically passes an integer between the running processes of different ranks and lets the
process with rank 0 reduce the value of the integer by one. When the value reaches 0 the loop
breaks and the application finishes. This example was compiled and tested on the system used
to assure that the SPMD client was running properly. The C-code for the example can be
found in the built directories of the used Open MPI (v1.5). A schematic view of the
application is presented in Figure 9.

![Schematic View of the Application](image)

Figur 9: Schematic view of the ring_c.c example. The rank 0 process initializes the value of the message to 10
and passes the message to the next process. The value is decremented by 1 for each lap. Once a process receives
message = 0 the loop will break and the process will exit, lastly rank 0 will exit.

It was also tested whether the default MPI engine could through the `mpirun` command, run
several instances of a compiled application created by the MATLAB Compiler. This was
necessary to investigate since it would show if running several instances of the MATLAB
Compiler Runtime (MCR) on the same system was possible, something which would be a
future dependency of the parallelized PopED. This was successful as well, although it took
some time for the system to fully launch the MCR-instances the first time.

The main issue with parallelizing PopED through the usage of Open MPI is the level of
programming required to achieve said goal. Instead of having an almost automated approach
as the MATLAB Parallel Computing Toolkit provides, the developer will now be dependent
on a predefined MPI library which must be implemented manually to create an manager of the
message passing and data communication, basically a job manager and worker client that can
scale well with an “infinite” number of clients. The main approach would be to take
advantage of the MATLAB Compiler and from the existing MATLAB functions that do the
FIM calculation, create a shared C/C++ library which the job manager/worker will be able to
call. This will also provide the developer with a high level of control since the implementation
requires a message passing and memory allocation defined by the developer rather than the
MPI library. The Open MPI libraries already contain commands for send and receive requests
required by the MPI, so most issues with inter-process communication can be solved by using
already existing classes and functions. The main advantages and disadvantages with using Open MPI to achieve a fully functional parallelized PopED are presented in table 3.

**Table 3: Advantages and Disadvantages with using Open MPI**

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>High level of control of parallelization</td>
<td>More complicated solution than MATLAB PCT</td>
</tr>
<tr>
<td>Most necessary MPI-functions are already predefined in OMPI</td>
<td>Installation of OMPI on HPC resources require both creating libraries and compiling - Complicated</td>
</tr>
<tr>
<td>Possibly higher performance than automated parallelization with MATLAB PCT</td>
<td>The achieved performance is highly dependent on the skill of the programmer.</td>
</tr>
<tr>
<td>Distributed Memory architecture will scale well on the available HPC resources</td>
<td>This solution requires compilation of MATLAB scripts to C/C++-shared library.*</td>
</tr>
<tr>
<td>Open Source and Free solution!</td>
<td>Solution requires MATLAB Compiler Runtime*</td>
</tr>
<tr>
<td>Required spmd.exe client is lightweight and available for most platforms</td>
<td>Requires parsing for communication between MATLAB and C.*</td>
</tr>
</tbody>
</table>

*See section 4.

**3.3 Other Considered MPI solutions.**

**MPICH2:** A somewhat simpler API than Open MPI with a native GUI for launching load-balanced applications. This could be useful if the entire optimization was hardcoded and compiled into one executable, which is not the case for PopED. It is quite a bit easier to manage and install than Open MPI since it is a precompiled binary installer for specific platforms. How this would perform together with PopED will depend on the level of parallelization, a strict process cloning might not achieve anything in this case. Both MPICH2 and Open MPI follow the MPI-2 standard and thus have similar commands and functionality when implemented. So if this library is of interest it might be better to simply use the MPI source of MPICH2 which is found in Open MPI as well. [11]

**OpenMP:** Another MPI solution not to be confused with the previously described Open MPI. The main difference between OpenMP and Open MPI is that it is developed by another board of experts forming the so called OpenMP Architecture Review Board (ARB). The ARB consists of permanent members, such as Microsoft, Intel and other hardware retailers and Auxiliary members from other less commercial areas such as NASA. OpenMP Uses a shared memory architecture and consist of compiler directives, run time routines and environment...
variables which have to be implemented by the programmer by using predefined “pragmas” in C/C++. If done in the correct manner the performance should be at the highest possible level, although this might be hard to achieve during this project since we want the parallelized PopED to be as versatile as possible by using distributed memory architecture. There is an extensive amount of documentation available for OpenMP including tutorials and publications. This does still remain an alternative for some systems, but at the moment it is not available on the clusters available for testing. The ultimate solution would be to combine both Open MPI and OpenMP for optimized local and distributed execution, although due to the strict time limit for this project and the fact that one of the available clusters only has Open MPI installed, OpenMP will for now not be implemented.[12,13]

4. Implementation of parallelization of PopED with Open MPI

After discussion with involved parties it was decided that the project would continue with parallelizing PopED by an implementation of Open MPI. Upon deciding on Open MPI a few main problems emerged which later had to be taken into consideration:

- At which level of the optimization script shall the parallelization be implemented?
- How shall the MATLAB Part communicate with the implemented C/C++ job manager/worker?
- How shall User Defined models in MATLAB be accessed by the worker?
- How to effectively handle data sending/receiving to avoid a queue of ready workers waiting to return data to the job manager?

The most straightforward approach at the time was to simply attack the problem at the population FIM calculation level and to take advantage of the MATLAB Compiler and its runtime to create a client which communicates with Open MPI and does the actual computational work. The schematic view of the suggested application addition to PopED is presented in Figure 10.
4.1 Changes in MATLAB functions

The first task was to add a FIM- calculation script by adding a function `calc_fim(...)` which receives the design variables from the Worker and then passes it on to the existing FIM calculation script to perform the FIM calculation. The full code for `calc_fim.m` can be found as Appendix B. This function was compiled and tested with hard-coded design parameters simply to check whether it was efficient enough to proceed with.

The first idea of having one compiled executable which acts as the parallel worker who does all the computational work of the FIM calculation was found to effectively stop the functionality of editing user defined models. It showed that a by MATLAB compiled application cannot access the generated .m-files (through the MATLAB Compiler Runtime) which contained the model functions and variables. This problem was countered by having the main function of PopED, `poped(...)`, call a new custom function, `compile_poped(...)`, which sets the variables and collects the necessary functions to send to `compile_fim(...)` function which compiles the FIM-calculation script, including the model functions, to a C-
shared library. This library includes a .dll (in a windows environment, .so in a UNIX environment) and the complete functionality of the MATLAB FIM-calculation script translated into C, and thus made accessible for parallelization with the C-library of Open MPI. This compilation is the first of two compilation steps required to run PopED in parallel. By calling the root function for FIM-calculation, calc_fim(...), the MATLAB Compiler automatically includes all the necessary dependencies (m-files) for the FIM-calculation. The second compilation links the created shared library and the MPI functionality and some other help-functions which are necessary for the full communication between MATLAB and C.

All common functions in PopED have access to a structure containing definitions and settings for the optimization, model and designs. Here this structure was updated to include settings for the compilation and parallel execution. The compilation setting globalStructure.parallelSettings.iCompileOptions with a default value of -1 was added. If this is changed to 1 or 2 it will call the compile_poped(...) function and compile according to the first or second compilation step described previously. By setting the variable globalStructure.parallelSettings.iCompileOptions equal to 0, PopED will perform both compilation steps. Since it is the latter which creates the executable needed for performing the parallel calculations both steps will have to be done before proceeding with the parallel execution. The value of globalStructure.parallelSettings.iCompileOptions can be changed manually by the user by setting the value of popedInput.parallelSettings.iCompileOptions found in the generated function_input() file.

After the compilation was implemented it was decided that the issue with communication between the source client and the working job manager/worker should be managed by two .mat files. Thus the designs to be evaluated were saved in .mat files using the default MATLAB function for .mat file creation. This is possible since the compilation process includes the libraries, libmx, libmat mclmcr or mclmcrrt, dependent on MATLAB version, which contains the .mat read/write functions in MATLAB when creating the shared library. The .mat-file for input to the worker-client has a default name of par_input.mat with the addition of a run-specific number (0 – 10^7) which was randomly generated. The results from the worker was saved in another .mat file, called par_output.mat with the same random run number corresponding to the par_input.mat file used for the FIM-calculation. The full name of the input-file is passed on to the worker client with the system call as one of the input parameters.
Other minor changes was made as well in the `doptim(...)` and `line_search_a(...)` functions, which consisted of saving the designs to be evaluated in cells structures which were passed on to the new function `execute_parallel(...)` instead of evaluating the designs one by one as in the serial approach. After receiving the design parameters, `execute_parallel(...)` generates the random run-specific number, creates the `par_input.mat`, and generates a string to use for the system-call which executes the parallel run. The function also calculates the size of the FIM which will be calculated and adds that to the string. When finished the system call string contains the initialization call to Open MPI, the number of processes, name of the executable, names of the input and output .mat-files, name of the `function_input()`, number of designs to send to the workers at a time (see section 4.2) and the length of the FIM row. The system call will then (on Windows) look similar to the example string below.

<table>
<thead>
<tr>
<th>Example 2: The system call for parallel execution of the FIM-calculation on a Windows OS</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mpirun -np 3 calc_fim.exe par_input123456.mat par_results123456.mat function_input 5 10 &amp;</code></td>
</tr>
</tbody>
</table>

The setting `–np 3` is simply a call for open MPI to initialize 3 instances of the process `calc_fim.exe` and the & sign is something necessary for the system to understand that this shall be run in the background, i.e. the executing system shell should not wait for `mpirun` to end. After the `execute_parallel(...)` function has launched a parallel execution with the system command it will enter a waiting loop which breaks when the function finds, and successfully parses, an `par_output.mat` file with the same ID-number as the recently launched process. When the waiting-loop breaks, `execute_parallel(...)` will finish and the `doptim(...)` or `a_line_search(...)` function, depending on which function that made the call, will extract the results from the generated `par_output.mat` (now stored as a cell-structure) and continue with the optimization script.
4.2 The Job Manager/Worker

Open MPI was implemented in C/C++ to achieve the message passing required for the parallel computations of the FIM. This part of the application, named `mpi_console1.cpp`, utilizes the finished function defined by the MPI-2.2 standard and found within the Open MPI libraries. It is `mpi_console1` which acts as the base for the job manager and worker processes by having two specific parts of the code with different functions and separated by a simple if-statement monitoring the rank of the process. By default the lowest rank, rank 0, is the designated job manager and all the above are workers. This implies that the user will have to consider that it is preferable to dedicate one core to the job manager process, which means that the minimum number of cores required by the application to effectively run the FIM-calculations in parallel will be 3, otherwise one worker-process might be “bottlenecked” by having the job manager running on the same core.
The MPI-2.2 standard already contain the commands for rank determination, which is executed for all initialized instances which then receives their rank as a process variable. The process will continue with its designated role by either entering the loop for the job manager or worker depending on the rank. The job manager and worker loop is described further in sections 4.2.1 and 4.2. All ranks have been programmed to read the designs to be evaluated from the par_input.mat file and saving these in a cell structure called designlist. The function is called getPopEDDesignStructuresFromMat(…) and utilizes the .mat read functions from the included MATLAB C-headers (mat.h, matrix.h, mex.h) and then creates the cell structure containing the designs. This function and several other functions, vital for the calculation process to work properly, are defined in mat_function.h. The full functionality of mat_functions is presented in section 4.2.3.

All the data containing communication between processes are managed by the commands MPI_Send(…) and MPI_Recv(…) with different message tags for the different messages. There are four different message types and message tags defined:

**INIT_TAG**, Tag with value 1 was implemented to indicate that the message content is the initialization message which contains the first job and status to the worker.

**STATUS_REP**, Tag with value 2 implemented to indicate that the message content is the status report message which is the worker’s way of telling the job manager that the results of its current batch of received jobs are ready to be returned to the job manager.

**OUTPUT_TAG**, Tag with value 3 implemented to indicate that the message content is the result of an FIM-calculation of one design that the worker wants to return to the job manager.

**NEW_STATUS**, Tag with value 4 implemented to indicate that the message content is the new status and job for a worker being sent from the job manager.

Furthermore there are three different types of data being sent between the workers and the job manager. It was done in this manner to minimize the weight of the data traffic on the total calculation time. The first idea was to have only two types of data packages, one for the status and one for the FIM-calculation result data, although this later was shown to be troublesome due to memory allocation conflicts. Thus the following data packages were implemented:

int job_message[4] = \{x,y,a,b\} \text{ where: } 0 \leq x,y \leq 1 \quad 0 \leq a < \text{numDesigns-1} \\
a \leq b < \text{numDesigns}
This short vector was implemented to act as the status message for both the first initialization and the new status sent from the job manager to the workers. The integer, $x$, is the variable monitored by the worker which indicates that the worker should still be active since there are still designs left to be evaluated. When a worker is about to be killed the first integer is changed to 0. The second integer, $y$, is the variable indicating that the worker is ready if the value is set to 1. The last two integers, $a$ and $b$, gives the offsets in the list of designs to be evaluated for which the worker is supposed to calculate the FIMs. The width of the interval, $[a-b]$ is set by a variable $calc\_block$ with a minimum value of 1. This is an input parameter used by the MATLAB system call as mentioned earlier. It was chosen to implement the interval in this manner instead of having fixed intervals since it makes it possible to have a dynamic range in case of one worker finishing faster than the other. Furthermore, by having $calc\_block$ as an input parameter it is possible for the user to easily tweak the interval by altering the PopED parallel setting variable $popedInput.parallelSettings.iNumChungDesignEvals$ found in the $function\_input.m$ file.

```plaintext
int result_message[2] = {a, b} 0≤a<numDesigns-1  a≤b<numDesigns
```

In essence a shortened version of the status vector only containing the $a$ and $b$ described above used to prepare the job manager to receive and sort the returning designs in the correct spot of the cell structure list containing results called $designoutlist$.

```plaintext
PopedOutputType designout = { MPI_INT designnr;
MPI_DOUBLE FIM [iFimDim][iFimDim];
MPI_DOUBLE ofv;
};
```

A committed (to the MPI environment) custom $MPI\_Datatype$ with its basic structure defined in the header file for $mpi\_console1.cpp$, $stdafx.h$. The structure is later applied to create the $MPI\_Datatype$ by allocating the MPI versions of the integers and doubles in the $designout$ MPI structure. This is later used by the worker to send the results to the job manager. The variable $iFimDim$ is an input parameter used to allocate the correct size of the FIM in the memory. It is received via the system call made by the MATLAB function $execute\_parallel(...)$.
The application also uses variable classes defined by the MATLAB headers (mex library) such as the `mxArray` class, which is the MATLAB compatible matrix format used to send parameters between C and MATLAB code.

### 4.2.1 Job manager

If the process variable `rank` has the value 0 the process will identify itself as the job manager and enter the segment of code dedicated to that purpose. The job manager first of all allocates the memory necessary for a number of parameters needed for the management of the worker processes, such as `rem_designs` and `sent_designs`, which are counters for the remaining number of designs and the number of sent designs respectively. Another important variable is `worker_ready`, this integer can have a value of 0 or 1 and acts as an indicator that there is a worker waiting to return results.

The variable `designoutlist`, which is used to store the results and later to create the `par_output.mat` file, is created by allocating memory with the standard `malloc`-function for the custom list class PopEDDesignoutStructure with the same length as the `designlist` variable. The size of the FIM within the structure is adjusted by using the `iFimDim` variable for all indices of the `designoutlist`.

The job manager then enters the first `for`-loop, which is the worker initialization loop that loops through the worker ranks and sends the first batch of jobs to the pool of workers. It alters the content of the `job_message` vector to initialize the workers by setting `x` and `y` to 1. Furthermore, the interval of designs to be evaluated is set by setting the `a` and `b` variable to the desired first and last design, which are stored by the local variables `from_des` and `to_des`. The job message is sent with the command below:

```c
MPI_Send(&job_message, 4, MPI_INT, i, INIT_TAG, MPI_COMM_WORLD);
```

Where `&job_message` is the reference to the job data to be sent, 4 is an indicator of how many of the datatype `MPI_INT` is supposed to be sent, `i` is the rank of the recipient, `INIT_TAG` is the message specific tag described earlier and `MPI_COMM_WORLD` is a variable used to indicate the MPI instance to which the message belong. The `MPI_Send(…)` is a blocking method which waits until the message has been received, which is necessary to avoid a message to being missed due to the worker not being ready to receive it. After the workers has received the first jobs and been fully initialized the job manager exits the initialization loop,
updates the `from_des` and `to_des` variables by adding the value of `calc_block` and starts looking for finished workers by using the status probing command below:

```c
MPI_Iprobe(MPI_ANY_SOURCE, STATUS_REP, MPI_COMM_WORLD, &worker_ready, &status);
```

This function checks for an incoming message from finished workers with the status report tag `STATUS_REP` and from any source by using the wildcard `MPI_ANY_SOURCE` as the source parameter. If a ready worker is found, the function will change the value of the parameter `worker_ready` to 1, thus allowing the job manager to enter the section of code dedicated to receiving results from the ready worker and saving them in the `designoutlist`. The function `MPI_Iprobe(...)` is a non-blocking function which simply checks if there are any incoming messages, and if there is none it simply continues. The reason for choosing this instead of having a blocking receive function directly at this level is to achieve an adaptive receive/send algorithm for the job manager. And since the top `while`-loop will repeat as long as `rem_designs` is greater than 0, i.e. there is still designs left to be received, the probe will be repeated until a ready worker has been found.

The result data receiving section of the job manager code has been implemented in such a manner that it is dependent on the three major variables `rem_designs`, `sent_designs` and `worker_ready`. The `worker_ready` variable is the main variable for occasion of denied entry to the receive section since it is dependent on the ready status of workers, although as the number of workers increase the chance for the probe function to find a ready worker will increase and the variable `sent_designs` might become more limiting. The variable `sent_designs`, which has its value increased as soon as a worker has work sent to it by the job manager, will limit the receive section to only be available if there is jobs left to send to the workers, i.e. `sent_designs` is less than `cellength`. If that statement is false the job manager will instead of entering the receive mode, proceed further by sending a kill message (x = 0) to the worker. The `worker_ready` variable is only changed back to 0 if a successful send has been performed.

If a worker ready to send the results are found the job manager will call a blocking receive of a message from the source which was declared ready with the following command:

```c
MPI_Recv(&result_message, 2, MPI_INT, status.MPI_SOURCE, STATUS_REP, MPI_COMM_WORLD, &status);
```
The receive function will change the value of the \textit{result\_message} to the interval of designs which has been evaluated by the ready worker. The rank of the source worker is set through the \textit{status} structure by accessing the field \textit{status.MPI\_SOURCE} and the tag of the message is \textit{STATUS\_REP}. This receive call is one of two different types used, the latter located within a \textit{for}-loop running from \texttt{a} to \texttt{b} to receive the results from the worker and saving the results in the \textit{designoutlist}. The only difference between these calls is that the data type to be received is set to \texttt{PopedOutputType} and that it alters the \texttt{designout} variables instead of the \textit{result\_message}. After it has received the data for one design the value of \texttt{rem\_designs} is decremented by one. When all the designs has been received and saved in the \textit{designoutlist} the while-loop will break and the job manager will continue to save the \textit{designoutlist} in by creating the \texttt{par\_output.mat} file with the content of the \textit{designoutlist}. This is done by calling the function \texttt{save\_design\_structure(...)} defined in \texttt{mat\_functions.cpp}. After the designs have been saved, the job manager sends out a kill message to all the workers which has been waiting for their next status message by setting the value of \texttt{job\_message[0]} to 0. Finally the job manager will terminate and the PopED MATLAB script will send back the calculated designs, saved in the \texttt{par\_output.mat} file, to the optimization algorithm.

\textbf{4.2.2 Worker}

The workers who do the actual computations are designed to be strictly controlled by the job manager. Thus following a worker startup algorithm, similar to the startup sequence of the job manager, is a blocking \texttt{MPI\_Recv(...)} which the workers cannot pass until they have received a message with the message tag \texttt{INIT\_TAG} from the job manager. The main difference in startup sequence between worker and job manager is that the workers initializes a shorter \textit{designoutlist} than the job manager and that they also initializes a connection to the MATLAB Compiler Runtime and the MATLAB C-shared libraries compiled earlier. After receiving this message, the workers will enter the calculation sequence which is a \texttt{while}-loop that repeats as long as the value of \texttt{job\_message[0]} is greater than 0. Finally, before entering the evaluation script, the workers check if the upper limit of the evaluation interval, given by \texttt{job\_message[3]}, is within the length of the \textit{designlist}. If that is not the case, the worker will adjust the upper limit to the length of the \textit{designlist}, i.e. by setting \texttt{job\_message[3]} to \texttt{celllength}.

The workers will evaluate the designs at offsets of the \textit{designlist} given by the received \texttt{job\_message} vector. The design parameters for design \texttt{i} are collected from \texttt{designlist[i]} and saved in a temporary design structure where each element is of the \texttt{mxArray} class, this is done
to ensure compatibility with the predefined MATLAB functions that are included. To perform the actual FIM-calculation the worker uses `mlfCalc_fim(...)` which calls the `calc_fim` function, compiled by MATLAB to a c-shared library. The output is saved in their version of `designoutlist` which has a shorter length than the full `designoutlist` that the job manager has, this was implemented simply to reduce the memory needed by the workers. After the worker `designoutlist` has been filled with finished evaluated designs the worker enters a blocking send to the job manager which reports that the worker is done and which designs that has been evaluated. This message has the tag `STATUS_REP` and contains the data `result_message` describe earlier. Once the job manager has received this message the worker will enter the for-loop which sends the design results in `designoutlist` to the job manager, one by one, with a blocking `MPI_Send()` with tag `OUTPUT_TAG` and the data `designout` described previously.

After a successful calculation and data return algorithm the worker enters a final blocking `MPI_Recv()` which expects a new `job_message` vector with tag `NEW_STATUS` from the job manager. If there is no more designs to be sent from the job manager to the workers, the workers will simply wait in this blocking receive until they eventually receive the kill message from the job manager which sets the value of `job_message[0]` to 0 and breaks the loop. When the workers have exited the while-loop which keeps them active they will simply close the connection to the MATLAB Compiler Runtime and shut the process down.

### 4.2.3 Additional Dependencies

As mentioned earlier, `mpi_console1.cpp`, uses functions defined by Open MPI and thus the headers and libraries from the Open MPI project is included into the `mpi_console1.cpp` build. Furthermore, headers and libraries defined in the MATLAB folders are used to access MATLAB specific functions and classes such as the `.mat` read/write functionality and the `mxArray` class. The FIM-calculation functions which have been compiled into a c-shared library by the MATLAB compiler are included as well. Finally additional functions required by `mpi_console1` where implemented in `mat_functions.cpp` and `help_functions.cpp`.

**`mat_functions.cpp`:** Contains the functions required to read the `.mat` files and save the design variables in list of structures accessible by the job manager/worker. It also contains functions to copy designs between structures and functions to print the designs in the command window for debug mode. Furthermore there are functions which are used to free the memory allocated for the designs by destroying each element in the initialized lists of designs. The function
which is called by the job manager to save the `designoutlist` in the par_output.mat file can be found here as well.

**help_functions.cpp:** Contains two small functions useful for debugging of a run. The function `print_debug()` is a custom printing algorithm which takes a debug level flag as input which can control whether the algorithm prints anything or not. This makes it possible by having that flag as a process variable to control the output of all `print_debug` commands without commenting them out in the code. The function `print_matrix` also takes one debug level flag as well as data and matrix dimensions and prints the data in a matrix with the specified dimensions. This function is used to print the FIM in the command window during debug mode.

Some adaptations of the `function_input(...) and execute_parallel(...) to fit the SLURM scheduler on the Kalkyl HPC-cluster has been made. These include a modified system call in `execute_parallel.m` which instead of directly calling `mpirun`, first calls the SLURM scheduler and reserves the nodes and cores given by `parallelSettings.iNumProcesses` for a time the user has to specify in `parallelSettings.strExtraRunOptions`. Both these are found in the `function_input`. The second part of this “system call” is a call to a `.sh` script which takes the parameters and launches the processes over the reserved cores when they are available with the `mpirun` command.

A chart showing the approximate workflow of the `mpi_console1.cpp` is presented in figure 12 and the full code for `mpi_console1.cpp` with dependencies can be found as Appendix C, or within the `compile_tools` folder amongst the PopED program folders.
Figur 12: A simplified view of the workflow within the job manager/worker processes. Intra process communication are shown as red arrows with the corresponding data and message tag above and below the arrow line respectively.

5. Testing
Available for testing the parallelization of PopED was the UPPMAX HPC-Cluster Kalkyl, which as of January 2011 had a total of 348 nodes with dual Intel quad core E5520 running at 2.27GHz and 24-72 GB of RAM per node. This adds up to a total of 2784 available cores with a total of 9504 GB of RAM. The nodes are interconnected with infiniband and the cluster has a distributed memory architecture with SLURM as the scheduler. The required software
for testing; MATLAB 7.10, Open MPI 1.4.2 and gcc 4.4.2 are available and can be loaded to each node with a system call.

Optimization of two different models where used for testing, one lightweight one-compartment model with first order absorption and one slightly more computational intense two-compartment model with first order absorption. Initial testing was done with first order (FO) approximation of FIM but after the stability was considered trustworthy, the approximation method was changed to FOCE to increase the computational load on the CPU’s during the performance evaluation.

The models were implemented in the PopED GUI on a local computer to generate the model files and function_input() which later were transferred to Kalkyl. The models are included in PopED’s built-in predefined models and were not modified, all parameter were set to their default value whilst allowing stable parameter estimates for the optimization. The parallel settings where implemented directly in the generated function_input(...) since the possibility to change parallel settings within the GUI was not yet implemented at the time.

Both models where first executed via the standard serial optimization script followed by a parallel execution with eight cores, i.e. seven workers and one job manager. Finally the models where evaluated with 31 workers and one job manager. The time required to perform the random searches and line searches when optimizing sampling times was recorded for both the one-compartment and the two-compartment model. The results are presented in Table 4 and Table 5 respectively.

To better evaluate the effectiveness of the parallel communication a third model was implemented, designed to be the most calculation intensive of the three models tested. This three compartment model with a zero rate infusion drug administration was run in serial and parallel with 8 and 32 processes. The results from these optimizations are presented in table 6. This model optimizes both sampling times and covariates.

The experimental speed increase, \( \delta_{exp.} \), is derived by dividing the average serial run time per FIM calculation with the corresponding parallel run time per FIM calculation accordingly:

\[
\delta_{exp.} = \frac{t_{\text{serial}}}{t_{\text{parallel}}}
\]  

(19)

The average effectiveness \( \Delta \) of the parallelization was evaluated by comparing the average experienced speed increase, \( \delta_{exp.} \), with the theoretical speed increase, \( \delta_{\text{theor.}} \):
\[ \Delta = \frac{\delta_{\text{exp.}}}{\delta_{\text{theor.}}} \]  

(20)

The theoretical speed increase is given by the number of workers evaluating the FIM.

Furthermore, the average time spent per FIM evaluation for message passing and worker initialization, \( t_{\text{MP}} \), was calculated according to Equation 21.

\[ t_{\text{MP}} = (1 - \Delta) \cdot t_{\text{FIM}} \]  

(21)

Where \( t_{\text{FIM}} \) is the average time required to calculate one FIM, including all message passing and initialization. It is derived by the total evaluation time divided by the number of evaluated designs. The \((1 - \Delta)\) can be considered the weight of the message passing relative to the speed of the FIM calculation. The results of the evaluation are presented in Table 7.

6. Results

The results of the optimizations where deemed reasonable and the data which were relevant for this project is presented below:

Table 4: Elapsed times for the serial and parallel random search and line search algorithms for different number of workers when optimizing sampling times for the one-compartment model. For parallel execution one of the cores are reserved to the job manager and thus does not perform any FIM-calculations.

<table>
<thead>
<tr>
<th></th>
<th>Serial Execution</th>
<th>8 Cores</th>
<th>32 Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Random Search</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RS times (s):</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(300 FIMs)</td>
<td>7058.52</td>
<td>7101.66</td>
<td>1191.98</td>
</tr>
<tr>
<td></td>
<td>7038.96</td>
<td>7070.37</td>
<td>1145.45</td>
</tr>
<tr>
<td><strong>Mean time (s):</strong></td>
<td>7067.38</td>
<td>1144.71</td>
<td>270.65</td>
</tr>
<tr>
<td><strong>t_{FIM} (s)</strong></td>
<td>23.56</td>
<td>3.816</td>
<td>0.902</td>
</tr>
<tr>
<td><strong>( \delta_{\text{exp.}} )</strong></td>
<td>N/A</td>
<td>( \approx 6.1 )</td>
<td>( \approx 26.1 )</td>
</tr>
<tr>
<td><strong>Line Search</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>LS times (s):</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(153 FIMs)</td>
<td>3579.27</td>
<td>3555.43</td>
<td>680.86</td>
</tr>
<tr>
<td></td>
<td>3602.07</td>
<td>3561.92</td>
<td>652.24</td>
</tr>
<tr>
<td><strong>Mean time (s):</strong></td>
<td>3574.67</td>
<td>658.58</td>
<td>207.54</td>
</tr>
<tr>
<td><strong>t_{FIM} (s)</strong></td>
<td>23.36</td>
<td>4.304</td>
<td>1.36</td>
</tr>
<tr>
<td><strong>( \delta_{\text{exp.}} )</strong></td>
<td>N/A</td>
<td>( \approx 5.3 )</td>
<td>( \approx 17.18 )</td>
</tr>
</tbody>
</table>
Table 5: Elapsed times for the serial and parallel random search and line search algorithms for different number of workers when optimizing sampling times for the two-compartment model. For parallel execution one of the cores are reserved to the job manager and thus does not perform any FIM-calculations.

<table>
<thead>
<tr>
<th></th>
<th>Serial Execution</th>
<th>8 Cores</th>
<th>32 Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Random Search</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RS times (s): (300 FIMs)</td>
<td>8830.68</td>
<td>8768.22</td>
<td>1370.24</td>
</tr>
<tr>
<td></td>
<td>8758.08</td>
<td>8905.54</td>
<td>1452.98</td>
</tr>
<tr>
<td>Mean time (s):</td>
<td>8786.88</td>
<td>1405.94</td>
<td>331.266</td>
</tr>
<tr>
<td>tFIM (s)</td>
<td>29.29</td>
<td>4.687</td>
<td>1.04</td>
</tr>
<tr>
<td>(\delta_{\text{exp.}})</td>
<td>N/A</td>
<td>(\approx 6.25)</td>
<td>(\approx 26.5)</td>
</tr>
<tr>
<td><strong>Line Search</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LS times (s): (255 designs)</td>
<td>7562.29</td>
<td>7255.84</td>
<td>1227.18</td>
</tr>
<tr>
<td></td>
<td>7318.73</td>
<td>7265.72</td>
<td>1223.53</td>
</tr>
<tr>
<td>Mean time (s):</td>
<td>7350.65</td>
<td>1227.17</td>
<td>331.83</td>
</tr>
<tr>
<td>tFIM (s)</td>
<td>28.82</td>
<td>4.812</td>
<td>1.3</td>
</tr>
<tr>
<td>(\delta_{\text{exp.}})</td>
<td>N/A</td>
<td>(\approx 5.99)</td>
<td>(\approx 22.2)</td>
</tr>
</tbody>
</table>

Table 6: Elapsed times for serial and parallel random search and line search algorithms for different number of workers when optimizing sampling times and covariates for the three-compartment model. For parallel execution one of the cores are reserved to the job manager and thus does not perform any FIM-calculations.

<table>
<thead>
<tr>
<th></th>
<th>Serial Execution</th>
<th>8 Cores</th>
<th>32 Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Random Search</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RS times (s): (300 FIMs)</td>
<td>11231.7</td>
<td>11252.9</td>
<td>1776.38</td>
</tr>
<tr>
<td></td>
<td>11216.3</td>
<td>11269.9</td>
<td>1806.54</td>
</tr>
<tr>
<td>Mean time (s):</td>
<td>11242.7</td>
<td>1789.16</td>
<td>425.448</td>
</tr>
<tr>
<td>tFIM (s)</td>
<td>37.48</td>
<td>5.964</td>
<td>1.418</td>
</tr>
<tr>
<td>(\delta_{\text{exp.}})</td>
<td>N/A</td>
<td>(\approx 6.28)</td>
<td>(\approx 26.43)</td>
</tr>
<tr>
<td><strong>Line Search</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LS times (s): (204 FIMs)</td>
<td>7490.78</td>
<td>7458.51</td>
<td>1212.00</td>
</tr>
<tr>
<td></td>
<td>7577.61</td>
<td>7503.06</td>
<td>1187.81</td>
</tr>
<tr>
<td>Mean time (s):</td>
<td>7507.49</td>
<td>1194.88</td>
<td>427.945</td>
</tr>
<tr>
<td>tFIM (s)</td>
<td>36.80</td>
<td>5.857</td>
<td>2.098</td>
</tr>
<tr>
<td>(\delta_{\text{exp.}})</td>
<td>N/A</td>
<td>(\approx 6.28)</td>
<td>(\approx 17.54)</td>
</tr>
</tbody>
</table>
7. Conclusion
The implemented parallelization has to be classified as successful, although not perfect. With
the average message passing weight on calculation power being slightly inconsistent, it was
hard to judge the effectiveness other than considering the raw computational power the extra
CPUs provide as the number of workers increase as an achieved positive effect. The overall
time for the search methods does decrease as the number of cores increase, which was the
main goal of the parallelization. However, the relative difference in speed between is
consistently lower for the line search algorithm, and especially in cases of higher number of
workers. This will have to be investigated and possibly, the line search will have to be re-
evaluated.

8. Discussion
The parallel version of PopED remains, at the time, in testing phase. There were some
stability issues connected with the initialization of the compiled C-shared library on several
workers simultaneously. Thus making workers sporadically crash with increased frequency as
the number of workers increase. When this occurred, it occasionally helped if the name of the
executable was changed in the function_input, all files generated by the compilation process
were deleted and the MCR cache folder was cleared. This might be related to a bug when the
MCR is loading its cache when launching compiled files which source are on another user
directory with restricted access. This sometimes leads to the MCR trying to read and write a
cache with another user account than the user which is running the application. The problem
was solved by adding an additional command –C to the compilation of the C-shared library
which then copies the cache to the run directory instead of using the cache under the user root directory.

It should be noted that all these problems only occurred on the Kalkyl HPC-cluster with a UNIX environment. On a PC with the Windows environment in which the parallelization was implemented, the parallel PopED is much more stable. However this might be due to the fact that, on a local computer, all MCR processes are running on the same machine, and not across several separate nodes.

Thus the main issue with this project has not been the implementation per se, but rather how to adapt the program, developed in a Windows environment, to the available UNIX clusters. Most of the time spent troubleshooting has been about how to adapt to the UNIX system and about random bugs emerging when scaling the number of workers from a desktop to a HPC-cluster.

It has also been discussed that to optimize the performance and compatibility of the parallel PopED it would be an idea to implement an OpenMP solution as well. This would increase performance on both local executions, i.e. on one node or desktop, and on a HPC-cluster with a shared memory architecture.

The overall implementation still has some room for improvement; it would also be of interest to evaluate the apparent speed increase in comparison with what could have been achieved by using the MATLAB Parallel Computing Toolkit together with equally many MATLAB Distributed Computing Server clients as there were workers during the testing. One other reason for parallelizing with the MATLAB PCT is to allow the user to run PopED in parallel over four cores, which is the maximum that the MATLAB PCT allows without the use of additional MATLAB licenses or the MATLAB DCS. This can be useful for the purpose of local execution of PopED on a regular multi-core computer, which might be desirable for testing of models before launching it full scale on a HPC-cluster. It should be relatively simple to implement now that PopED is parallelized with Open MPI by simply using the existing added code for parallel execution and replacing the compilation and call to an external process with a simple `parfor`-loop around the serial FIM-calculation script `mf_tot(...)`. Furthermore, only linear search and a non adaptive random search have been fully parallelized, thus leaving the other algorithms PopED utilizes in serial. This was done due to limited time for the project. It should be mentioned that parallelizing e.g. the modified Fedrov
exchanged algorithm, \texttt{mfea(...)}, would be almost as easy as it was to parallelize the linear search algorithm \texttt{a_line_search(...)}. But since the stochastic gradient is used in collaboration with the random search and Line search, the next algorithm to parallelize will be SG to finally have the most common “search package” complete, which will allow optimization of more advanced models than those used during the testing.

Regarding the experienced performance of the implemented parallelization, it is apparent that the message passing and process initialization is the main factor for reduced performance. In theory, as the time required to calculate one FIM increases, the less the message passing will affect the overall performance. It was expected that the weight of the message passing on the entire evaluation time would continuously decrease as the FIM calculations became more intense, although not proved here; it is still likely that it can be proved by further testing and with more computational intense models. The testing was run on the same nodes for the one and two compartment, but during the testing with the three compartment model, the Kalkyl HPC-cluster was under more pressure by other users, which might have affected the performance of the test.

The experienced performance increase is seemingly less for the Line Search implementation. In the third model this might be explained by the LS algorithm being divided into different sections where the sampling times, the covariates and the discrete variables are being optimized in parallel, one after another. This leads to the line search being more dependent on message passing since, if all three optimizations are done, there will be three separate calls to the parallel Job Manager/Worker client. One solution, which would reduce the waiting time as the system launches the parallel processes, would be to implement the parallelization in such a manner that it only requires one system call for all three parameters types.

All the testing was done with the default value of the parameter \texttt{iNumChunkEval}, 5. This was done to simply increase the message passing to better evaluate the stability and the weight of the message passing. If this parameter would have its value increased to its maximum, to i.e. a value which leads to the total number of designs being evenly divided across all workers with one job batch, the weight of the message passing should reduce and the apparent effectiveness should increase. This is especially valid for cases where each FIM is computationally light. However, this means that the dynamic job distribution will be void, which means that the slowest machine will bottleneck the entire calculation. This is possible for clusters which have been gradually expanded with hardware newer than the previous installation.
Finally, the parallelization of PopED is not fully completed. What has been implemented has to be tested further and possibly changed to increase performance, and the methods MFEA and SG still has to be implemented. But in all the future for a fully parallel PopED is, with some work, bright.

In other words, the optimizer has to be optimized.

9. Acknowledgements
I want to thank all my supervisors during this project, especially Joakim Nyberg, whose utmost patience with teaching me the basics in how the PopED software is structured was vital to this project. And even though he has been off on a paternity leave, he still had the time to help me with the debugging process.

Finally I want to thank my dear fiancée Caroline, who has endured all the evenings she has spent alone as I have been working extra with the project to achieve the desired results.

10. References


Appendix:

A: parForTestSweep.m
B: calc_fim.m
C: mpi_console1.cpp
function [speedUp]=parForTestSweep(O,P)
matlabpool open 2;% can adjust according to your resources (in this case, a
dualcore processor)

M = 200;
disp('This function will generate a number of matrices containing the maximum eigenvalues for N different 200x200 randomized matrices');
disp(['The number of iterations will sweep between ' num2str(O) '-' num2str(P) '.']);
timeSer =[];
timePar =[];
numberOfIterations=[];
speedUp=[];
for N=O:P;
tSer=tic;
a=[];
b=[];
% serial (regular) for-loop
for i = 1:N
    a = [a max(eig(rand(M)))];
end

time1=toc(tSer);
timeSer=[timeSer time1];

tPar = tic;
% parallel for-loop
parfor j = 1:N
    b = [b max(eig(rand(M)))];
end

time2 =toc(tPar);
timePar=[timePar time2];
numberOfIterations = [numberOfIterations N];
speedUp=[speedUp time1/time2];
end;

matlabpool close;
%plots the elapsed time for each N
figure;
plot(numberOfIterations,timeSer,numberOfIterations,timePar,numberOfIterations,speedUp);
title('Time(Number Of Iterations)');
xlabel('Number Of Iterations');
ylabel('Time to completion of For-loop');
legend('Serial Loop (s)','Parallel Loop (s)','SpeedUp (parallel/serial)');
end
Appendix B

function [designvar] = calc_fim(xtin,ain,xin,grpszin,niin,designnr,strInputName,iDebugMode,iGroup)

if (iDebugMode>1)
    fprintf('Start Calculating FIM number %d...\n',designnr);
end

%Read the popedInput structure
if (iDebugMode>1)
    fprintf('Reading in the PopED input structure %s...\n',strInputName);
end
inp=eval(strInputName);

%creates the global structure, the container of parameters, by importing it
%from the PopEd generated files
globalStructure=convert_popedInput(inp);
globalStructure= convert_variables(globalStructure);

if (iDebugMode>1)
    fprintf('Downsizing general design\n');
end

[ni, xt, model_switch, x, a, bpop, n, d, maxxt,minxt,maxa,mina] = downsizing_general_design(globalStructure);

%Checking for non-default design variables
if (iDebugMode>1)
    fprintf('Checking input variables\n');
end
if (~isempty(xtin) || grpszin > -1 || niin > -1 || xin > -1 || ain>-1)
    if (~isempty(xtin))
        xt = xtin;
    end
    if (grpszin > -1 )
        globalStructure.groupsize = grpszin;
    end
    if (niin > -1)
        ni = niin;
    end
    if (xin > -1)
        x = xin;
    end
    if (ain > -1)
        a = ain;
sigma = diag(globalStructure.sigma);
d_full = getfulld(d(:,2),globalStructure.covd);
docc_full = getfulld(globalStructure.docc(:,2),globalStructure.covdocc);

if (iDebugMode>1)
    fprintf('Design');
end

if (iDebugMode>1)
    fprintf('Designs variables when calculating FIM ...
');
    disp(xt);
end

if (~isempty(iGroup) && iGroup>0)
    fprintf('Calculating FIM with mf_all ...
');
    mf_all(model_switch(iGroup,1:ni(iGroup))',xt(iGroup,1:ni(iGroup))',x(iGroup,:)',a(iGroup,:)',bpop(:,2),d_full,sigma,docc_full,gl obalStructure);
else

if (globalStructure.d_switch)
    [FIM,globalStructure]=mftot(model_switch,globalStructure.groupsize,ni,xt,x,a,bpop(:,2),d_full,sigma,docc_full,globalStructure);
    %calculate ofv
    detFIM=ofv_fim(FIM,globalStructure);
else

    [FIM,detFIM,globalStructure]=ed_mftot(model_switch,globalStructure.groupsize,ni,xt,x,a,bpop,d,globalStructure.covd,sigma,globalS tructure.docc,globalStructure);
end
end

% structure containing the design parameters and determinant of FIM
designvar = {};
designvar.designnr = designnr;
designvar.a = a;
designvar.xt = xt;
designvar.groupsize = globalStructure.groupsize;
designvar.ni = ni;
designvar.x = x;
designvar.fim = FIM;
designvar.detFim = detFIM;
if (iDebugMode>1)
    fprintf('Done Calculating FIM number %d\n',designnr);
end
end
Appendix C

// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
// are changed infrequently
//
#pragma once

#include <stdio.h>
#include <iostream>

#pragma once
#include <mpi.h>  //MPI functions
#include "calc_fimC.h"  //Matlab functions
#include "mat.h"
#include "matrix.h"
#include "mex.h"
#include "mat_functions.h"  //Helper functions

struct DesignOutStructure
{
    int designnr;  //Design number
    double *FIM;   //Fisher Information matrix
    double ofv;   //Objective function value
};

static const double POPED_DEBUG=2;  //The debug level 0 - No Debug, 1 - Medium, 2 - Full output
#define TMP_FILENAME "results.mat"  //The default output name if none is specified
// mpi_console1.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"
#define INIT_TAG 1
#define STATUS_REP 2
#define OUTPUT_TAG 3
#define NEW_STATUS 4

int main(int argc, char* argv[])
{
    //---------------------------------------
    int iFimDim = atoi(argv[5]);

    int rank, size;
    mxArray *cell=NULL;
    mxArray *out = NULL;
    double n=100, m=200;
    double xt[2] = {0.2, 0.7};
    double groupsize = -1;
    double ni = -1;
    double x = -1;
    double designnr = 1;
    double a = -1;

    //---------------------------------------

    unsigned int calc_block = atoi(argv[4]);
    unsigned int result_message[2] = {0,0};
    unsigned int job_message[4] = {0,0,0,0};
    unsigned int cellength = 0;
    unsigned int ret_data = 0;
    MPI_Status status;
    mxArray *mxxt,*mxa,*mxgroupsize,*mxni,*mxx,*mxdesignnr,*mxdebug,*mxstrInputFunction, *mxiGroup;
    PopEDDesignStructure * designlist;
    PopEDDesignOutStructure * designoutlist;
    DesignOutStructure *designout;

    print_debug(2,"Initialize matlab application\n");
    /* Call the MCR and library initialization functions */
if(!mclInitializeApplication(NULL,0))

{  
    print_debug(0,"Could not initialize the application.
    exit(1);
}

print_debug(2,"getting poped designs structures...

desiglist = GetPopEDDesignStructuresFromMat(argv[1],cellength);
print_debug(2,"%d poped designs structures collected from %s...

cellength,argv[1]);

MPI_Datatype PopEDOutputType;
MPI_Datatype type1[3] = {MPI_INT,MPI_DOUBLE, MPI_DOUBLE};
int blocklen1[3] = {1,iFimDim*iFimDim, 1};
MPI_Address( designout, disp1);
MPI_Address( designout[0].FIM, disp1+1);
MPI_Address( &designout[0].ofv, disp1+2);
base1 = disp1[0];
for (int i=0; i < 3; i++)
{
    disp1[i] -= base1;
}
MPI_Type_struct( 3, blocklen1, disp1, type1, &PopEDOutputType);
MPI_Type_commit(&PopEDOutputType);

//----------MPI initialization-----------------

print_debug(2,"MPI initialization...

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
print_debug(2,"MPI initialization done for Rank %d

//----------End of MPI initialization---------

//----------Job Manager-----------------------

designout = (struct DesignOutStructure *)malloc(sizeof(struct DesignOutStructure));
designout[0].FIM = (double*) malloc(sizeof(double)*iFimDim*iFimDim);
MPI_Address( designout, disp1);
MPI_Address( designout[0].FIM, disp1+1);
MPI_Address( &designout[0].ofv, disp1+2);
base1 = disp1[0];
for (int i=0; i < 3; i++)
{
    disp1[i] -= base1;
}
MPI_Type_struct( 3, blocklen1, disp1, type1, &PopEDOutputType);
MPI_Type_commit(&PopEDOutputType);

//----------Job Manager-----------------------

if (rank==0) {
    //message to workers from rank 0. {workleftswitch, gotworkforyouswitch, from_des, to_des}
    int worker_ready = 0;
    int result = 0;
    unsigned int from_des = 0;
    unsigned int to_des = calc_block;
    unsigned int rem_designs = cellength;

    unsigned int sent_designs = 0;
    designoutlist = (PopEDDesignOutStructure *)malloc(sizeof(PopEDDesignOutStructure) * cellength);
    for (int k = 0; k < cellength; k++)
        designoutlist[k].FIM = (double*) malloc(sizeof(double) * iFimDim * iFimDim);

    //Initialization of workers
    for (unsigned int i = 1; i < size; i++)
    {
        print_debug(2, "Initializing Worker %d\n", i);
        job_message[0] = 1;
        job_message[1] = 1;
        job_message[2] = from_des;
        job_message[3] = to_des;

        MPI_Send(&job_message, 4, MPI_INT, i, INIT_TAG, MPI_COMM_WORLD);
        print_debug(2, "Sent designs %d-%d to Worker %d\n", from_des + 1, to_des, i);
        sent_designs = sent_designs + (to_des - from_des);
        from_des = from_des + calc_block;
        to_des = to_des + calc_block;
    }

    while (rem_designs >= 0)
    {
        //testing for Incoming messages
        MPI_Iprobe(MPI_ANY_SOURCE, STATUS_REP, MPI_COMM_WORLD, &worker_ready, &status);

        if (rem_designs > 0 && worker_ready == 1)
        {
            print_debug(2, "result Message[1-2] = %d - %d. Found ready worker: %d.\n", result_message[0], result_message[1], status.MPI_SOURCE);

            result_message[0] = 0;
            result_message[1] = 0;

            MPI_Recv(result_message, 2, MPI_INT, status.MPI_SOURCE, STATUS_REP, MPI_COMM_WORLD, &status);

            print_debug(2, "Recieved ready-status from worker %d with designs %d-%d.\n",
                        status.MPI_SOURCE, result_message[0] + 1, result_message[1]);
        }
    }
for (unsigned int z = result_message[0]; z < result_message[1]; z++)
{
    print_debug(2,"Jobmanager is receiving designresult %d from rank %d\n", z+1, status.MPI_SOURCE);
    MPI_Recv(designout, 1, PopEDOutputType, status.MPI_SOURCE, OUTPUT_TAG, MPI_COMM_WORLD, &status);
    print_debug(2,"Jobmanager has received designresult %d from rank %d\n", z+1, status.MPI_SOURCE);

    print_debug(2, "Saving ofv %d in Designoutlist\n", z+1);
    designoutlist[z].ofv = designout->ofv;
    print_debug(2, "ofv %d has been saved in Designoutlist\n", z+1);
    print_debug(2, "Saving FIM %d in Designoutlist\n", z+1);
    memcpy(designoutlist[z].FIM, designout->FIM, sizeof(double)*iFimDim*iFimDim);
    print_debug(2, "FIM %d has been saved in Designoutlist\n", z+1);

    rem_designs--;
}

if (sent_designs < cellength && worker_ready == 1)
{
    //if the remaining number of designs to send is more than the block_length used
    job_message[2] = from_des;
    job_message[3] = to_des;
    if (to_des <= cellength)
    {
        MPI_Send(&job_message, 4, MPI_INT, status.MPI_SOURCE, NEW_STATUS, MPI_COMM_WORLD);
        sent_designs = sent_designs + (to_des - from_des);
        to_des = to_des + calc_block;
    }
    else //if the remaining number of designs to send is less than the block_length used
    {
        to_des = cellength;
        print_debug(2,"from_des : %d to_des: %d\n", from_des, to_des);
        job_message[3] = to_des;
    }

    print_debug(2,"Sending last jobs to rank %d\n", status.MPI_SOURCE);
    MPI_Send(&job_message, 4, MPI_INT, status.MPI_SOURCE, NEW_STATUS, MPI_COMM_WORLD);
    print_debug(2,"Sent designs %d to Worker %d\n", from_des +1, to_des, status.MPI_SOURCE);
    sent_designs = sent_designs + (to_des - from_des);
}

worker_ready = 0;
}

if (rem_designs <= 0)
{
    break;
print_debug(2,"Copying Designs to DesignOutlist\n");
for (unsigned int i = 0; i<cellength; i++)
{
    PopEDDesignStructure design = designlist[i];
    designoutlist[i].design = CopyDesign(&design);
    designoutlist[i].fimsize_dim = iFimDim;
}

// Part that saves the data into an output .mat file...

print_debug(2,"Saving results to MAT-file\n");
const char *filename=TMP_FILENAME;
if (argc>2)
filename = argv[2];

if (save_design_structure(filename,designoutlist,cellength)!=0)
    print_debug(0,"Error writing results file %s\n",filename);

for (int k = 1; k< size; k++)
{
    job_message[0] = 0;
    print_debug(2,"Sending kill Message to rank %d\n",k);
    MPI_Send(&job_message,4,MPI_INT,k,NEW_STATUS,MPI_COMM_WORLD);
}

print_debug(2,"Jobmanager is Freeing PopED structures in DesignOutlist...\n");
FreePopEDDesignOutStructures(designoutlist,cellength);
FreePopEDDesignStructures(designlist,cellength);
print_debug(2,"Jobmanager is done and shutting down\n");

} //end if rank==0
else{
    //----------------------------------------------------------------------------WORKER----------------------------------------------------------------------------

    int eval_counter = 0;

designoutlist=(PopEDDesignOutStructure *)malloc(sizeof(PopEDDesignOutStructure)*calc_block);
for (int k=0; k<calc_block; k++)
    designoutlist[k].FIM = (double*) malloc(sizeof(double)*iFimDim*iFimDim);
if (cellength>0)
{
    print_debug(2,"%d designs are in the memory of rank %d\n",cellength,rank);
}

    /* Do the Matlab work */
if(!calc_fimCInitialize())
{
    print_debug(0,"Could not initialize the library.\n");
    exit(1);
}

MPI_Recv(&job_message, 4, MPI_INT, 0, INIT_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

while (job_message[0] == 1)
{
    if (job_message[3] > cellength){
    }
    if (job_message[1] == 1)
    {
        print_debug(2,"Message recieved in rank %d is: %d %d %d %d\n", rank, job_message[0], job_message[1], job_message[2], job_message[3]);

        unsigned int j = 0;

        //Calc a fim for each design in the mat file
        //message[2] = from design with index
        //message[3] = to design with index
        for (unsigned int i=job_message[2]; i<job_message[3]; i++)
        {
            mxArray *output=NULL;
            PopEDDesignStructure design = designlist[i];
            mxxt=mxCreateNumericMatrix(design.m,design.maxni,mxDOUBLE_CLASS,mxREAL);
            memcpy(mxGetPr(mxxt),design.xt, mxGetElementSize(mxxt)*design.m*design.maxni);
            mxa=mxCreateNumericMatrix(design.m,design.na,mxDOUBLE_CLASS,mxREAL);
            memcpy(mxGetPr(mxa),design.a, mxGetElementSize(mxxt)*design.m*design.na);
            mxx=mxCreateNumericMatrix(design.m,design.nx,mxDOUBLE_CLASS,mxREAL);
            memcpy(mxGetPr(mxx),design.x, mxGetElementSize(mxxt)*design.m*design.nx);
            mxgroupsize=mxCreateNumericMatrix(design.m,1,mxDOUBLE_CLASS,mxREAL);
            memcpy(mxGetPr(mxgroupsize),design.groupsize, mxGetElementSize(mxxt)*design.m);
            mxn=mxCreateNumericMatrix(design.m,1,mxDOUBLE_CLASS,mxREAL);
            memcpy(mxGetPr(mxn),design.ni, mxGetElementSize(mxxt)*design.m);

            double iter = (double)design.iter;
design.iGroup = 0;
double iGroup = (double)design.iGroup;
mxiGroup=mxCreateNumericMatrix(1,1,mxDOUBLE_CLASS,mxREAL);
memcpy(mxGetPr(mxiGroup),&iGroup, mxGetElementSize(mxxt));

if (mlfCalc_fim(1,
&output,mxxt,mxa,mxx,mxgroupsize,mxni,
mxdesignnr,mxstrInputFunction,mxdebug,mxiGroup)==false)
{
  print_debug(0,"Could not run Calc_fim application!\n");
  exit(1);
}

if (output!=NULL)
{

designoutlist[j].design = CopyDesign(&design);

mxArray *cell = output;
mxArray *mxfim = mxGetField(cell, 0, "fim");
mxArray *num = mxGetField(cell, 0, "designnr");
mxArray *mxofv = mxGetField(cell,0,"detFim");

size_t ndim = mxGetNumberOfDimensions(mxfim);
const mwSize *dims = mxGetDimensions(mxfim);

designoutlist[j].fimsize_dim = (int)dims[0];
designoutlist[j].FIM= (double*)malloc((int)dims[0]*(int)dims[1]*mxGetElementSize(mxxt));

memcpy(designoutlist[j].FIM,mxGetPr(mxfim),
(int)dims[0]*(int)dims[1]*mxGetElementSize(mxfim));
memcpy(&designoutlist[j].ofv,mxGetPr(mxofv),mxGetElementSize(mxofv));

  eval_counter++;
  j++;
};//end for
//changes status to waiting for work
result_message[0]=job_message[2];
result_message[1]=job_message[3];

print_debug(2,"rank 1 is sending status message: %d %d\n",
    result_message[0], result_message[1]);

MPI_Send(&result_message, 2, MPI_INT, 0, STATUS_REP, MPI_COMM_WORLD);
for (unsigned int i = 0; i < (job_message[3] - job_message[2]); i++)
{
    memcpy(designout->FIM, designoutlist[i].FIM, sizeof(double)*iFimDim*iFimDim);
    designout->ofv = designoutlist[i].ofv;
    print_debug(2,"Rank %d is sending designresult %d to Jobmanager\n",
        rank, (result_message[0] + i + 1));
    MPI_Send(designout, 1, PopEDOutputType, 0, OUTPUT_TAG, MPI_COMM_WORLD);
    print_debug(2,"Rank %d has sent designresult %d to Jobmanager\n",
        rank, (result_message[0] + i + 1));
}
job_message[1] = 0;
}//end if got work
else{
    print_debug(2,"Rank %d is waiting for work...",rank);
}
MPI_Recv(&job_message, 4, MPI_INT, 0, NEW_STATUS, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}//end while work_left

print_debug(2, "Rank %d has recieved kill message from Job manager\n", rank);
FreePopEDDesignStructures(designlist,cellength);
print_debug(2, "Rank %d has killed the design list\n", rank);

print_debug(2, "Rank %d is terminating detfimC\n", rank);
calc_fimCTerminate();

ret_data = eval_counter;

print_debug(2,"Rank %d is deleting designoutlist!\n",rank);
FreePopEDDesignOutStructures(designoutlist,calc_block);
print_debug(2,"Worker of rank %d has been killed!\n",rank);
}
print_debug(2,"Rank %d is Freeing PopED designout...\n",rank);
print_debug(2,"Rank %d is terminating Application\n",rank);
mclTerminateApplication();
MPI_Type_free(&PopEDOutputType);
print_debug(2,"Rank %d is finalizing MPI...
",rank);

MPI_Finalize();
print_debug(2,"Rank %d is going home...
",rank);

return 0;
}

#ifndef _MAT_FUNCTIONS_H
#define _MAT_FUNCTIONS_H
#include "help_functions.h"

struct PopEDDesignStructure
{
    int m;           //Num groups
    int maxni;       //Max num samples
    int na;          //Num a
    int nx;          //Num x
    double *xt;      //Samples
    double *a;       //Covariates
    double *x;       //Discrete vars
    double *groupsize; //Groupsize per group
    double *ni;      //Num samples per group
    int iGroup;      //flag for groupwise fimcalc
    int iter;        //Internal design iteration / number
};

struct PopEDDesignOutStructure
{
    PopEDDesignStructure *design;
    int fimsize_dim;  //The dimensions of FIM (5 means a 5x5 matrix)
    double *FIM;     //Fisher Information matrix
    double ofv;      //Objective function value
};

/* Function declarations */
mxArray * GetPopEDCellFromMat(const char *file, unsigned int &length);
PopEDDesignStructure *GetPopEDStructFromCell(mxArray *pa, int iter);
PopEDDesignStructure *GetPopEDDesignStructuresFromMat(const char *file, unsigned int &length);

void print_design_structure(const PopEDDesignStructure *design);

void FreePopEDDesignStructure(PopEDDesignStructure *design);
void FreePopEDDesignStructures(PopEDDesignStructure *designlist, unsigned int length);
void FreePopEDCell(mxArray *pa);
void FreePopEDDesignOutStructures(PopEDDesignOutStructure *designoutlist, unsigned int length);

PopEDDesignStructure *CopyDesign(PopEDDesignStructure *design);

void print_design_structure(const int iLevel, const PopEDDesignStructure *design);

int save_design_structure(const char *filename, PopEDDesignOutStructure *evaldesigns, const int numevals);

endif

#include "stdafx.h"
#include "mat_functions.h"

/* Open mat file containing poped design structure cell array, return the first mxArray */
mxArray *GetPopEDCellFromMat(const char *file, unsigned int &length) {
    MATFile *pmat;
    const char **dir;
    const char *name;
    int ndir;
    int i;
    mxArray *pa, *cell;
    length = 0;
    cell = NULL;

    /*
    * Open file to get directory
    */
    pmat = matOpen(file, "r");
    if (pmat == NULL) {
        print_debug(0,"Error opening file %s\n", file);
        return NULL;
    }
    print_debug(2,"File %s is opened..\n..\n",file);

    /*
* get directory of MAT-file
* /
dir = (const char **)matGetDir(pmat, &ndir);
if (dir == NULL) {
    print_debug(0,"Error reading directory of file %s\n", file);
    return NULL;
} else {
    print_debug(2,"Directory of %s:\n", file);
    for (i=0; i < ndir; i++)
        print_debug(2,"%s\n", dir[i]);
} mxFree(dir);

/* In order to use matGetNextXXX correctly, reopen file to read in headers. */
if (matClose(pmat) != 0) {
    print_debug(0,"Error closing file %s\n", file);
    return NULL;
}
pmat = matOpen(file, "r");
if (pmat == NULL) {
    print_debug(0,"Error reopening file %s\n", file);
    return NULL;
}

/* Get headers of all variables */
for (i=0; i < ndir; i++) {
    pa = matGetNextVariableInfo(pmat, &name);
    if (pa == NULL) {
        print_debug(0,"Error reading in file %s\n", file);
        return NULL;
    }
    mxDestroyArray(pa);
}

/* Reopen file to read in actual arrays. */
if (matClose(pmat) != 0) {
    print_debug(0,"Error closing file %s\n",file);
    return NULL;
}
pmat = matOpen(file, "r");
if (pmat == NULL) {
    print_debug(0,"Error reopening file %s\n", file);
    return NULL;
}

/* Read in each array. */
for (i=0; i<ndir; i++) {
    pa = matGetNextVariable(pmat, &name);
    if (pa == NULL) {
        print_debug(0,"Error reading in file %s\n", file);
        return NULL;
    }
    if (mxIsCell(pa)) {
        size_t size_m = mxGetM(pa);
        size_t size_n = mxGetN(pa);
        length = (unsigned int)size_n;
        if (matClose(pmat) != 0) {
            print_debug(0,"Error closing file %s\n",file);
            return NULL;
        }
        return pa;
    }
    else {
        print_debug(0,"mat file error - mat file element is not a cell\n");
    }
    mxDestroyArray(pa);
}
if (matClose(pmat) != 0) {
    print_debug(0,"Error closing file %s\n",file);
    return NULL;
}
print_debug(2,"Done reading file...");
return NULL;

void FreePopEDCell(mxArray *pa)
{
    if (pa!=NULL)
        mxDestroyArray(pa);
}

void FreePopEDDesignStructure(PopEDDesignStructure *design)
{
    free(design->xt);
    free(design->a);
free(design->x);
free(design->groupsize);
free(design->ni);
free(design);
}

void FreePopEDDesignStructures(PopEDDesignStructure *designlist, unsigned int length)
{
    for (unsigned int i=0; i<length; i++)
        FreePopEDDesignStructure(&designlist[i]);
    free(designlist);
}

void FreePopEDDesignOutStructures(PopEDDesignOutStructure *designoutlist, unsigned int length)
{
    for (unsigned int i=0; i<length; i++)
    {
        //if (designoutlist[i].design!=NULL)
        //    FreePopEDDesignStructure(designoutlist[i].design);
        //    
        //if(designoutlist[i].FIM != NULL){
        //    free(designoutlist[i].FIM);
        //}
    }
    free(designoutlist);
}

PopEDDesignStructure *GetPopEDStructFromCell(mxArray *cell, int iter)
{
    mxArray *celliter;
    mxArray *mxxt,*mxnum,*mxa,*mxx,*mxgroupsize,*mxni;
    size_t ndim;
    const mwSize *dims;
    // double *xt,*a,*x,*groupsize,*ni;

    PopEDDesignStructure *design;
    celliter = mxGetCell(cell,iter);
    design = (PopEDDesignStructure *)malloc(sizeof(PopEDDesignStructure));

    if (cell!=NULL)
    {
        if (celliter!=NULL && iter>=0)
        {

free(design->x);
free(design->groupsize);
free(design->ni);
free(design);
mxxt = mxGetField(celliter, 0, "xt");
mxa = mxGetField(celliter, 0, "a");
mxx = mxGetField(celliter, 0, "x");
mxgroupsize = mxGetField(celliter, 0, "groupsize");
mxni = mxGetField(celliter, 0, "ni");
mxnum = mxGetField(celliter, 0, "num");

//print_debug(0,"%s\n","Assigning dimensions to PopED design structure...");

ndim = mxGetNumberOfDimensions(mxxt);
dims = mxGetDimensions(mxxt);

design->m = (int)dims[0];
design->maxni = (int)dims[1];

dims = mxGetDimensions(mxa);
design->na = (int)dims[1];

dims = mxGetDimensions(mxx);
design->nx = (int)dims[1];

design->iter = (int)mxGetScalar(mxnum);


design->xt = (double *)malloc(design->m*design->maxni*sizeof(double));
design->a = (double *)malloc(design->m*design->na*sizeof(double));
design->x = (double *)malloc(design->m*design->nx*sizeof(double));
design->groupsize = (double *)malloc(design->m*sizeof(double));
design->ni = (double *)malloc(design->m*sizeof(double));

memcpy(design->xt,mxGetPr(mxxt),design->m*design->maxni*sizeof(double));
memcpy(design->a,mxGetPr(mxa),design->m*design->na*sizeof(double));
memcpy(design->x,mxGetPr(mxx),design->m*design->nx*sizeof(double));
memcpy(design->groupsize,mxGetPr(mxgroupsize),design->m*sizeof(double));
memcpy(design->ni,mxGetPr(mxni),design->m*sizeof(double));

return design;
}
else
{
    print_debug(0,"Couldn't get cell number %d\n",iter);
    return NULL;
}
return NULL;
else
    return NULL;
}

PopEDDesignStructure *CopyDesign(PopEDDesignStructure *design)
{
    PopEDDesignStructure *outdesign = (PopEDDesignStructure *)malloc(sizeof(PopEDDesignStructure));
    outdesign->m = design->m;
    outdesign->maxni = design->maxni;
    outdesign->na = design->na;
    outdesign->nx = design->nx;
    outdesign->iter = design->iter;

    outdesign->xt = (double *)malloc(outdesign->m*outdesign->maxni*sizeof(double));
    outdesign->a = (double *)malloc(outdesign->m*outdesign->na*sizeof(double));
    outdesign->x = (double *)malloc(outdesign->m*outdesign->nx*sizeof(double));
    outdesign->groupsize = (double *)malloc(outdesign->m*sizeof(double));
    outdesign->ni = (double *)malloc(outdesign->m*sizeof(double));

    memcpy(outdesign->xt,design->xt,outdesign->m*outdesign->maxni*sizeof(double));
    memcpy(outdesign->a,design->a,outdesign->m*outdesign->na*sizeof(double));
    memcpy(outdesign->x,design->x,outdesign->m*outdesign->nx*sizeof(double));
    memcpy(outdesign->groupsize,design->groupsize,outdesign->m*sizeof(double));
    memcpy(outdesign->ni,design->ni,outdesign->m*sizeof(double));

    return outdesign;
}

void print_design_structure(const int iLevel, const PopEDDesignStructure *design)
{
    print_debug(iLevel,"Design number: %d\n",design->iter);

    print_debug(iLevel,"Number of groups: %d\n",design->m);
    print_debug(iLevel,"Number of max samples: %d\n",design->maxni);
    print_debug(iLevel,"Number of covariates: %d\n",design->na);
    print_debug(iLevel,"Number of discrete variables: %d\n",design->nx);

    print_matrix(iLevel,"xt",design->xt,design->m,design->maxni);
    print_matrix(iLevel,"a",design->a,design->m,design->na);
    print_matrix(iLevel,"x",design->x,design->m,design->nx);
    print_matrix(iLevel,"groupsize",design->groupsize,design->m,1);
    print_matrix(iLevel,"ni",design->ni,design->m,1);
}

PopEDDesignStructure *GetPopEDDesignStructuresFromMat(const char *file, unsigned int &length)
PopEDDesignStructure *designlist;

mxArray *cell = GetPopEDCellFromMat(file,length);

if (cell==NULL) {
    print_debug(0, "%s\n", "Couldn't get PopED Cell");
    return NULL;
}

if (length==0) {
    print_debug(0, "%s\n", "No designs to evaluate!");
    return NULL;
}

print_debug(0, "Allocating memory for %d design structures...\n", length);
designlist=(PopEDDesignStructure *)malloc(sizeof(PopEDDesignStructure)*length);

print_debug(0, "Reading all %d design from mat file \%s...\n", length, file);

for (unsigned int i=0; i<length; i++)
    designlist[i]=*GetPopEDStructFromCell(cell,i);

FreePopEDCell(cell);

return designlist;

int save_design_structure(const char *filename,PopEDDesignOutStructure *evaldesigns,const int numevals) {
    MATFile *pmat;
    mxArray *cell;
    const char *fieldnames[3] = {"FIM","ofv","iter"};

    int status;
    print_debug(2,"Creating mat file \%s...
", filename);
    pmat = matOpen(filename, "w");

    if (pmat == NULL) {
        print_debug(0,"Error creating file \%s\n", filename);
        return -1;
    }

    cell = mxCreateCellMatrix(1, numevals); //Create the basic out structure cell

for(int i=0; i<numevals; i++)
{
    mxArray * structCell = mxCreateStructMatrix(1, 1, 3, fieldnames); //Create the struct
    mxArray * FIM = mxCreateDoubleMatrix(evaldesigns[i].fimsize_dim,
       evaldesigns[i].fimsize_dim, mxREAL);
    mxArray * ofv = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxArray * iter = mxCreateDoubleMatrix(1, 1, mxREAL);

    memcpy(mxGetPr(FIM), evaldesigns[i].FIM,
       evaldesigns[i].fimsize_dim*evaldesigns[i].fimsize_dim*sizeof(double));

    memcpy(mxGetPr(ofv), &evaldesigns[i].ofv,
       sizeof(double));
    memcpy(mxGetPr(iter), &evaldesigns[i].design->iter,
       sizeof(double));

    mxSetField(structCell, 0, fieldnames[0], FIM);
    mxSetField(structCell, 0, fieldnames[1], ofv);
    mxSetField(structCell, 0, fieldnames[2], iter);

    mxSetCell(cell, i, structCell);
}

status = matPutVariable(pmat, "designsout", cell);
if (status != 0)
{
    print_debug(0,"Error putting %s in mat file %s\n","designs",filename);
    return -1;
}

/* clean up */
mxDestroyArray(cell);
if (matClose(pmat) != 0) {
    print_debug(0,"Error closing file %s\n",filename);
    return -1;
}
return 0;
#ifndef _HELP_FUNCTIONS_H
#define _HELP_FUNCTIONS_H
#include <varargs.h>

/* Define the functions */
void print_matrix(const int iLevel, const char *name, const double *mat, const int m, const int n);
void print_debug(const int iLevel, const char * str, ...);
#endif

#include "stdafx.h"
#include "help_functions.h"

void print_debug(const int iLevel, const char * str,...)
{
    va_list argptr;
    if (iLevel<=POPED_DEBUG)
    {
        va_start(argptr,str);
        vprintf(str, argptr);
        va_end(argptr);
    }
}

void print_matrix(const int iLevel, const char *name, const double *mat, const int m, const int n)
{
    if (iLevel<=POPED_DEBUG)
    {
        printf("%s:\n",name);
        if (m==0 || n==0)
        {
            printf("\tempty (%dx%d -matrix)\n",m,n);
            return;
        }
        for (int i=0; i<m; i++)
        {
            for (int j=0; j<n; j++)
                printf("\t%g",mat[n*i+j]);
            printf("\n");
        }
    }
}