SFU SIMON FRASER UNIVERSITY
THINKING OF THE WORLD

Introduction to Optimum Design

MSE 426/726 Lab Manual

First Version: January 20, 2011
Latest Revision: January 5, 2021

G. [redacted]n.D., P. Eng., Professor
Mechatronic Systems Engineering
Simon Fraser University

Table of Contents

1 Introduction to MATLAB

1.1 Background of MATLAB

Originally developed to be a "matrix laboratory" by Cleve Moler, the recent versions, written in C by the MathWorks Inc., have capabilities far beyond the original MATLAB. It is an interactive system and programming language for general scientific and technical computation. There are over 350 numeric and graphical functions available in the recent versions of MATLAB. Associated with it are several so-called toolboxes, each providing a fairly large set of additional commands that are of particular use for computing tasks in a specific engineering area such as control, system identification, digital signal processing, neural networks, optimization, and so on. Together the MATLAB/Toolboxes package offers the user a powerful computing tool with superb graphics capabilities. More importantly, as MATLAB commands are similar to the way we express solution steps in mathematics, programming in MATLAB is much easier than in C or Fortran.

1.2 The MATLAB System

1. The MATLAB language. This is a high-level matrix/array language with control flow statements, functions, data structures, input/output, and object-oriented programming features. It allows both "programming in the small" to rapidly create quick and dirty throw-away programs and "programming in the large" to create complete, large, and complex application programs.

2. The MATLAB working environment. This is the set of tools and facilities that you work with as the MATLAB user or programmer. It includes facilities for managing the variables in your workspace and importing or exporting data. It also includes tools for developing, managing, debugging, and profiling M-files and MATLAB's applications.

3. Handle Graphics. This is the MATLAB graphics system. It includes high-level commands for two-dimensional and three-dimensional data visualization, image processing, animation, and graphical presentation. It also includes low-level commands that allow you to fully customize the appearance of graphics as well as to build complete Graphical User Interfaces on your MATLAB applications.

4. The MATLAB mathematical function library. This is a vast collection of computational algorithms ranging from elementary functions like sum, sine, cosine, and complex arithmetic to more sophisticated functions like matrix inverse, matrix eigenvalues, Bessel functions, and fast Fourier transforms.

5. The MATLAB Application Program Interface (API). This is a library that allows you to write C and Fortran programs that interact with MATLAB. It include facilities for calling routines from MATLAB (dynamic linking), calling MATLAB as a computational engine, and for reading or writing MAT-files.

1.3 Getting Started with MATLAB

In UNIX workstation, by entering **matlab** with the keyboard you should quickly see the MATLAB prompt (>>) waiting for you to enter a command. In PC workstation, run the MATLAB program by selecting it from the **start → program → MATLAB** menu, or double clicking on the **matlab** icon.

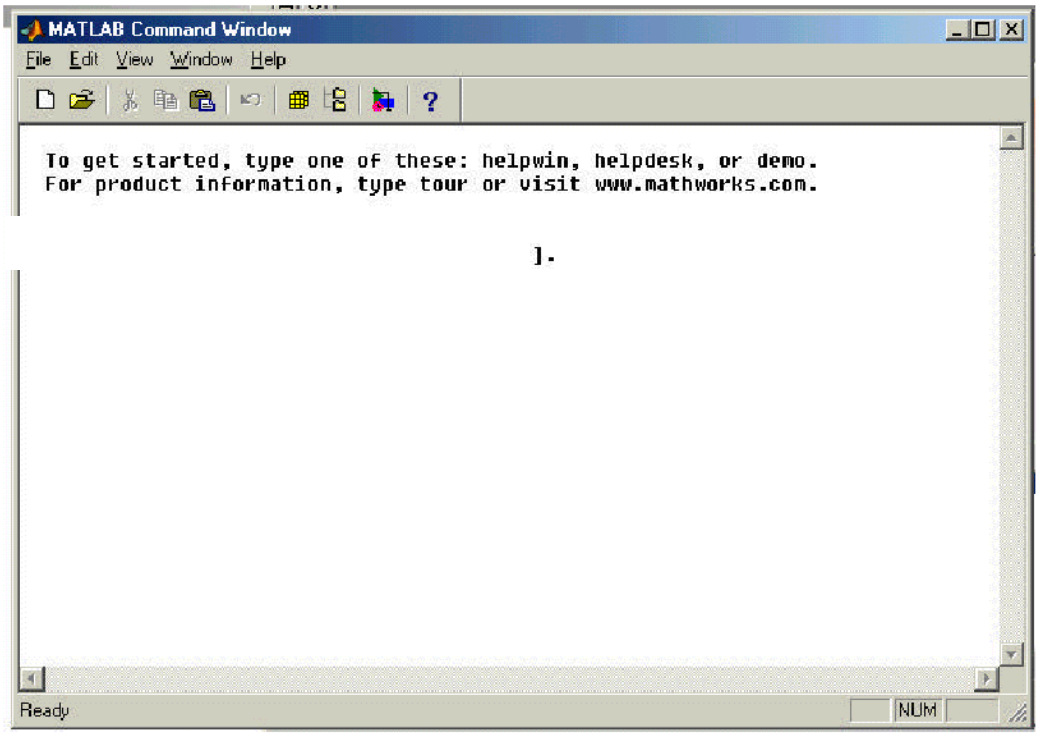

Figure 1-1 Matlab command window.

1.4 Working Modes of MATLAB

MATLAB provides two working modes. The first is the command mode, in which users type a command following the prompt (>>), and MATLAB executes the command right after pressing "Enter". In this mode, data is stored in the memory and is accessible unless being deliberately deleted (see command **clear** below). This command mode is useful for simple calculations and for learning or testing a command. Another working mode, the script mode, is to write a script file with suffix **.m**. One edits a script file by arranging commands in a sensible way. After saving the file, e.g., by the name **filename.m**, one can type the **filename** directly after the prompt (>>), and MATLAB will interpret and execute

the commands in the script file one by one. This mode is often used to develop programs or large projects that can be saved for future use.

1.5 Basic Commands

In what follows we attempt to identify and introduce a set of MATLAB commands that are used very often.

- If you need to know how to use a specific command, say **plot**, use command **help plot**. If you don't know the command or want to know more commands in a topic, use command **help** to see a list of help topics. You can then select the topic for which you want to have further information. The topics will be listed in accordance with the licenses that your lab may have for different modules and toolboxes. All of the fundamental topics are listed below:

  HELP topics:

matlab/general	-	General purpose commands.
matlab/ops	-	Operators and special characters.
matlab/lang	-	Programming language constructs.
matlab/elmat	-	Elementary matrices and matrix manipulation.
matlab/elfun	-	Elementary math functions.
matlab/specfun	-	Specialized math functions.
matlab/matfun	-	Matrix functions - numerical linear algebra.
matlab/datafun	-	Data analysis and Fourier transforms.
matlab/polyfun	-	Interpolation and polynomials.
matlab/funfun	-	Function functions and ODE solvers.
matlab/sparfun	-	Sparse matrices.
matlab/graph2d	-	Two dimensional graphs.
matlab/graph3d	-	Three dimensional graphs.
matlab/specgraph	-	Specialized graphs.
matlab/graphics	-	Handle Graphics.
matlab/uitools	-	Graphical user interface tools.
matlab/strfun	-	Character strings.
matlab/iofun	-	File input/output.
matlab/timefun	-	Time and dates.
matlab/datatypes	-	Data types and structures.
matlab/demos	-	Examples and demonstrations.
toolbox/tour	-	MATLAB Tour

  For more help on directory/topic, for instance, if you want to know all the matrix functions, type **help matfun** to view all the related commands.

- To change current directory, one can use **cd** in the same way as in a UNIX or DOS window.
- To exit MATLAB, use **quit** or **exit**.
- If you want to save some of the quantities that you have generated using MATLAB, say you want to save two matrices **A** and **B,** as well as a vector **c** as a data file named data-01 for future use, use the following command,

  save data-01 A B c

  After typing the above command, you will then find a file entitled **data-0l.mat** in your directory. Next time when you invoke the software again, use **load data-01** to load the data you saved.
- To abort a command in MATLAB, hold down the control key and then press the letter **c**.
- MATLAB is case-sensitive. For example, the variables named dt, Dt, and DT are different from each other.
- MATLAB uses a command window to enter comments and data and to print results, and uses a graphics window to display plots. Use **clf** to clear the graphics window. To clear variables, say A and c, use command **clear A c**. Executing command **clear** only (without any specified variables following it) will delete all the variables generated.
- If one adds a semi-colon at the end of a command, MATLAB will not generate any output for that command. Otherwise, the output will be printed to the screen after the command. For example, if one types **A=3+4;** after the prompt (>>), MATLAB will generate no output, and one will see another prompt after the command. If only **A=3+4** is typed, MATLAB will output A=7 on the screen. This feature is very useful in the script mode for users to control the program output.
- By default, the angle unit in MATLAB is radian. MATLAB also has pre-defined constant, **pi**, which can help you transfer the degree unit to radian.
- The command **who** lists the variables that you have defined, while command **whos** lists the variables along with their sizes.
- One can output a figure created by MATLAB to a file or a printer. For instance, assume that one has generated a plot. In the command line, type **print('filename.jpg', '-djpeg'),** and MATLAB will export the plot to the file **filename.jpg**. A variety of file types, besides the jpeg file, can be generated in the same way. For detailed info, please use **help print**.
- If you want to find more about Matlab, use its **Help → Product Help** or **Help → Demos** for quick learning.

1.6 Basic Operations

In this section we introduce more MATLAB commands for basic computations**.**

1.6.1 Arithmetic operations

Example: Calculate (1+2)*(1+2)/(3-2)*2.

The command in MATLAB will be:

```
>>(1+2)^2/(3-2)*2
18
>>
```

1.6.2 Matrix operations

1. Define a matrix.

```
>> A = [1 2; 2 3; 3 4]
A =
     1     2
     2     3
     3     4
```

Find the transpose of matrix A.

```
>>A'
ans =
     1   2   3
     2   3   4
```

2. Check the size of a matrix.

```
>> size(A)
ans =
     3     2
>>size(A, 2)
ans =
     2
```

The latter command refers to the number of columns of A. If we type size(A, 1), then the output will be 3, i.e., the number of rows.

3. Generate a matrix with all zeros and of the same size as A.

```
>> B = zeros(size(A))
B =
     0     0
     0     0
     0     0
```

4. Matrix reference

To refer to the data on the third row and the second column in A, the command is:
>> A(3, 2)
To refer to the first two rows of A:
>> A(1:2,:)
To refer to the second column of A:
>> A(:, 2)

5. Matrix operations

Matrix operations take the same form as basic arithmetic operations provided that all the matrices in the operations are of the same size, i.e., the same number of rows and columns.

1.6.3 Graphics Commands

- The most common graphics command is **plot**. Please type **help plot** in the MATLAB's command window to see the detailed description of the command.
- One can use **hold on** if multiple plots are expected to be printed in the same figure. Otherwise, use **hold off**. The default setting of MATLAB is **hold off**.
- One can use the **axis** command to control the output data range. For example, Axis([-2 2 –3 3]) specifies that the x-coordinate between [-2 2] and the y-coordinate between [-3 3] will be printed.
- One can use **title('your title')** to add a title to a plot.
- Commands **xlabel('your x label')** and **ylabel('your y label')** add label to a 2-D plot.

1.7 Matlab Toolboxes

MATLAB features a family of application-specific solutions called *toolboxes*. Very important to most users of MATLAB, toolboxes allow you to *learn* and *apply* specialized technology.

Toolboxes are comprehensive collections of MATLAB functions (M-files) that extend the MATLAB environment to solve particular classes of problems. Areas in which toolboxes are available include:

· signal processing,
· control systems,
· optimization,
· neural networks,
· fuzzy logic,
· wavelets,
· simulation,
· and many others.

For this course, three toolboxes are of relevance: "Optimization," "Global Optimization," and "Statistics." The next two chapters will talk about the first two toolboxes, which you will use for your lab assignments. Then a new tool, **Optimization Assisted Simulation Integration Software (OASIS)**, developed by the Empower Operations Corp. will be introduced.

Lab 1 Get Familiar with Matlab

Tasks:

1. Review the material written above; practice these commands when needed
2. Write a script to do the following:
   a. Plot the polynomial $(t) = t^4 - 3.5t^3 - 2.5t^2 + 14t - 6$ from t = -2.5 to 4.
   b. Add a label on the x-axis, a label on the y-axis, a title, and a grid. Also change the color of the graph to something other than blue.
   c. Use the data cursor to estimate all of the roots of the polynomial
   d. Save all your data into a file and export your plot into a JPEG file
3. Write a script to do the following:
   a. Code the following two functions as a subroutine. The input should be a point represented by a vector, and the output should be the function value.
   b. Generate plots and compare the plots with the graphs shown below
   c. Save the code and your plots

1) Keane's Bump Function

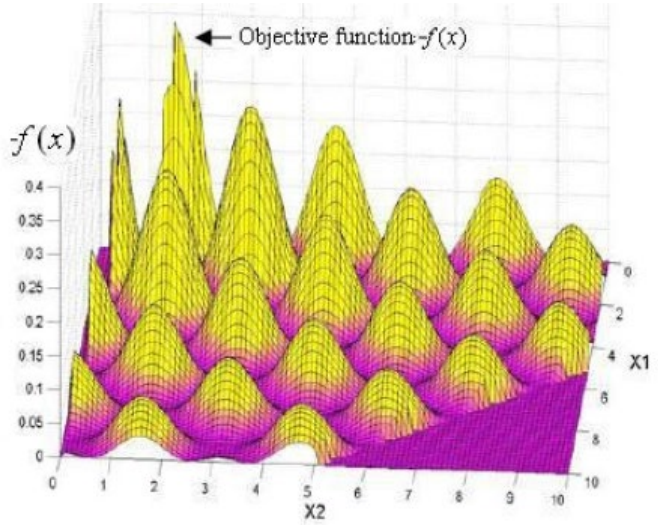

Figure 1-2: 2D Keane's Bump Function [1]

$$f(x) = -\left|\left[\sum_{i=1}^{d}\cos^4(x_i) - 2\prod_{i=1}^{d}\cos^2(x_i)\right] / \left(\sum_{i=1}^{d} i x_i^2\right)^{0.5}\right|$$

$$x \in [0, 10]; d = \#\ variables = 5$$

2) Shubert Function

$$f(\boldsymbol{x}) = \prod_{i=1}^{d}\sum_{j=1}^{5} j\cos\big((j+1)x_i + j\big)$$

$$x \in [-5.12, 5.12]; d = \#\ variables = 10$$

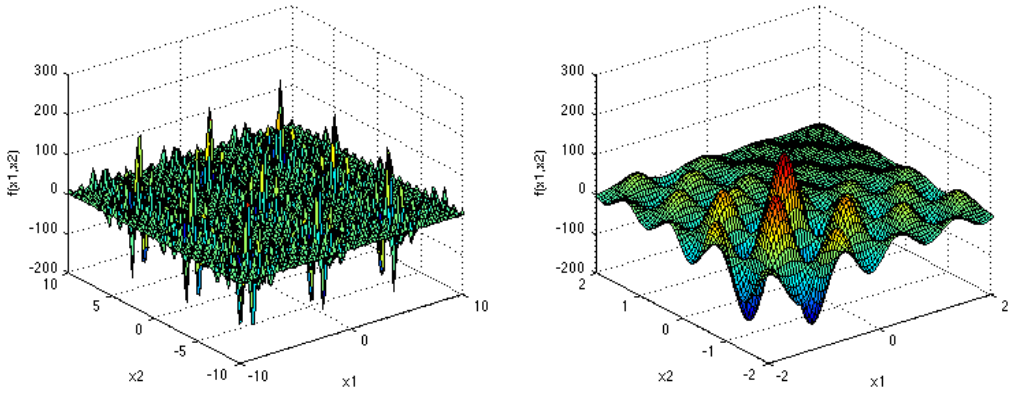

Figure 1-3: 2D Shubert Function [2]

4. Show your results and code to TA on screen

2 Introduction to the Matlab Optimization Toolbox

2.1 Start Using the Optimization Toolbox

The Optimization Toolbox consists of functions that perform minimization/maximization on linear/nonlinear constrained/non-constrained objective functions. These routines usually require the definitions of the objective functions in M-files. Alternatively, a string variable containing a MATLAB expression, with *x* representing the independent variables can be used. Optional arguments to the routines change optimization parameters and place bounds on the variables.

It is recommended that all files specific to the MATLAB environment be kept in a separate directory. Let's get started with the actual use of the Optimization Toolbox. After creating the new folder that will contain M-files and MAT-files and switching to that directory, MATLAB can be executed.

Of interest is the HELP command that is self-explanatory. You can get more information about the subject by entering:

>>help optim

Alternatively, one can find more demos and explanation by selecting **Help → Demos** and scrolling down to find Optimization, as shown in the next figure.

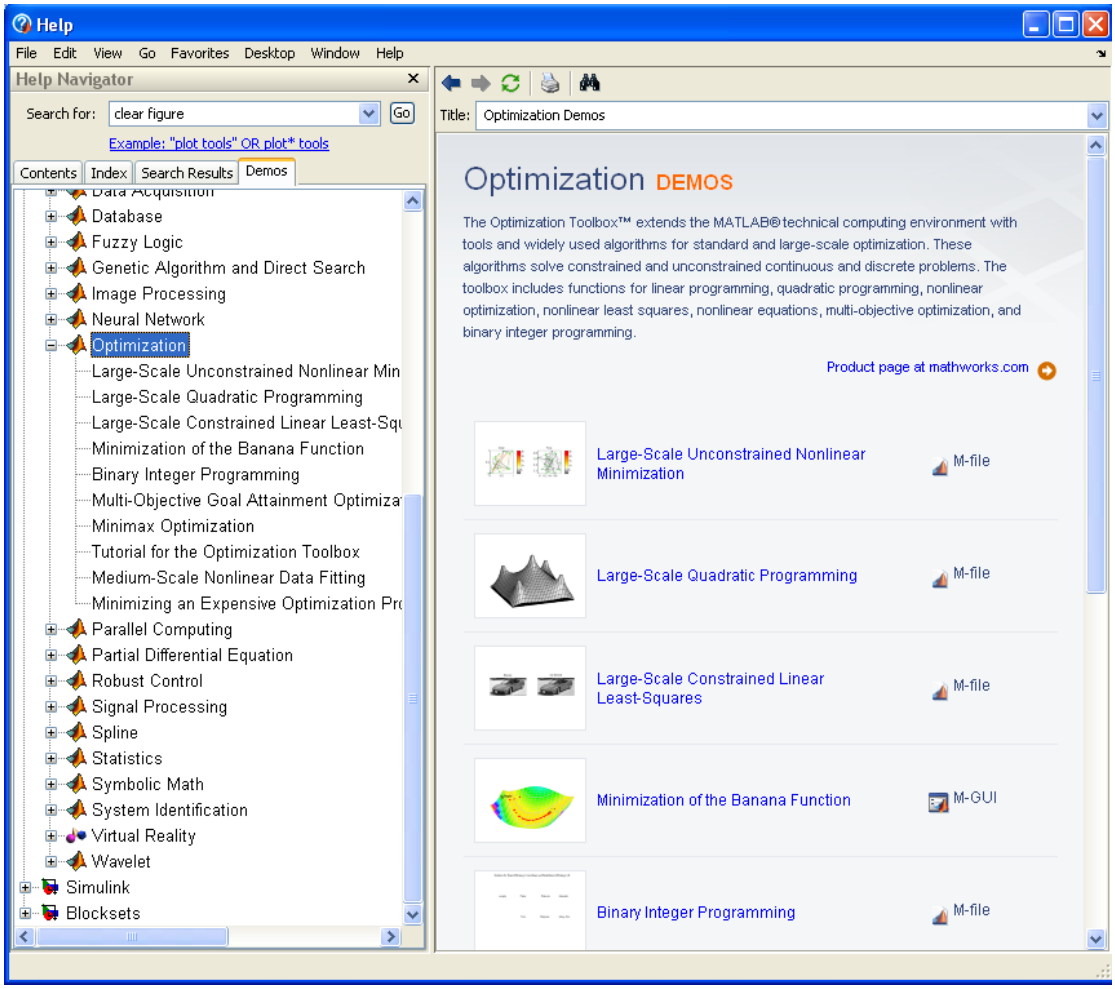

Figure 2-1 Finding demos from Matlab.

2.2 *Selection of Optimization Algorithms*

MATLAB Optimization Toolbox separates "medium-scale" algorithms from 'large-scale" algorithms. *Medium-scale* is not a standard term and is used here only to differentiate these algorithms from the *large-scale* algorithms, which are designed to handle large-scale problems efficiently.

2.2.1 Medium-Scale Algorithms

- The Optimization Toolbox routines offer a choice of algorithms and line search strategies.
  - The principal algorithms for unconstrained minimization are the *Nelder-Mead simplex search method* and the *BFGS quasi-Newton method*.
  - For constrained minimization, *minimax*, *goal attainment*, and *semi-infinite optimization*, variations of *Sequential Quadratic Programming* are used.
  - Nonlinear least squares problems use the *Gauss-Newton* and *Levenberg-Marquardt* methods.

- A choice of *line search* (or 1-D search) strategy is given for unconstrained minimization and nonlinear least squares problems. The line search strategies use safeguarded *cubic interpolation* (with analytical gradient functions) and *quadratic interpolation and extrapolation* (without analytical gradient functions) methods.

2.2.2 Large-Scale Algorithms

All the large-scale algorithms, except linear programming, are *trust-region* methods.

- Bound constrained problems are solved using *reflective Newton methods*.
- Equality constrained problems are solved using a *projective preconditioned conjugate gradient iteration*.
- You can use sparse iterative solvers or sparse direct solvers in solving the linear systems to determine the current step. Some choice of preconditioning in the iterative solvers is also available.

2.2.3 Linear Programming Problems

The *linear programming* method is a variant of *Mehrotra's predictor-corrector* algorithm, a *primal-dual interior-point* method.

2.3 Use of MATLAB Optimization Routines

1) Standard Form of the Optimization Problem

In order to use the optimization routines, the formulated optimization problem needs to be converted into the standard form required by these routines (case dependent).

2) Definition of Objective and Constraint Functions

Most of these optimization routines require the definition of an *M-file* containing the function to be minimized. The M-file, named *objfun.m,* returns the function value.

The constraints are specified in a second *M-file*, *confun.m*, that returns the value of the constraints at the current *x* in vector *c*.

The constrained minimization routine is then invoked. Maximization is achieved by supplying the routines with *-f*, where *f* is the function being optimized.

3) Optimization Parameter Setting

Optimization *options* passed to the routines change optimization parameters. Default optimization parameters are used extensively but can be changed through an *options* structure. Optimization *options* allow you to:

· select "medium-scale" or "large-scale" algorithms,
· select what kind of output to be displayed,
· set tolerance value, etc.

help optimoptions provides information that defines the different parameters and describes how to use them.

The initial search point, x0, has to be given by the user, based upon the problem.

4) Gradient Calculations

Gradients are calculated using an *adaptive finite-difference method* unless they are supplied in a function. Analytical expressions of the gradients of objective and constraint functions can be incorporated through gradient functions, G and DC, and proper *options* setting.

Parameters can be passed directly to functions, avoiding the need for global variables.

2.4 Optimization Toolbox Functions

Minimization

Function (Multidimensional)	***Purpose***
fgoalattain	Multiobjective goal attainment
fminbnd	Scalar nonlinear minimization with bounds
fmincon	**Constrained nonlinear minimization**
fminimax	Minimax optimization
fminsearch, **fminunc**	Unconstrained nonlinear minimization by Nelder-Mead direct search method
fseminf	Semi-infinite minimization
linprog	Linear programming
quadprog	Quadratic programming

Equation Solving

Function	***Purpose***
\	Linear equation solving (see the online *MATLAB Function Reference* guide)
fsolve	Nonlinear system of equations solving
fzero	Scalar nonlinear zero finding

Least-Squares (Curve Fitting)

Function	***Purpose***
\	Linear least squares (see the online *MATLAB Function Reference* guide)
lsqlin	Constrained linear least squares

lsqcurvefit	Nonlinear curve fitting via least squares (with bounds)
lsqnonlin	Nonlinear least squares with upper and lower bounds
lsqnonneg	Nonnegative linear least squares

Utility

Function	***Purpose***
optimoptions, optimget	Parameter setting

2.5 Examples

Find values of x that minimize $f(x) = -x_1x_2x_3$, starting at the point $x = [10; 10; 10]$, subject to the constraints:

$$0 \le x_1 + 2x_2 + 2x_3 \le 72.$$

1. Write an M-file that returns a scalar value f of the objective function evaluated at x. Save this function as myfun.m. (Note: "myfun" can be defined as you wish, as long as it is consistent with the one in the main file).

```
function f = myfun(x)
f = -x(1) * x(2) * x(3);
```

2. Rewrite the constraints as both less than or equal to a constant:

$$-x_1-2x_2-2x_3\le 0$$
$$x_1 + 2x_2 + 2x_3\le 72$$

3. Since both constraints are linear, formulate them as the matrix inequality $A\cdot x \le b$, where,

$$A=\begin{bmatrix}-1 & -2 & -2\\ 1 & 2 & 2\end{bmatrix},\ b=\begin{bmatrix}0\\ 72\end{bmatrix}.$$

4. Supply a starting point and invoke an optimization routine; the main function looks like below. Save the file as myrun.m.

```
x0 = [10; 10; 10];    % Starting guess at the solution
A=[-1 -2 -2; 1 2 2];
b=[0; 72];
[x,fval] = fmincon(@myfun,x0,A,b)
```

5. With the two files "myrun.m" and "myfun.m," saved in the current working directory, you can enter "myrun" in the command line. You should see the solution as below:

```
x =
```

```
    24.0000
    12.0000
    12.0000
```

where the function value is:

```
fval =
    -3.4560e+03
```

You can type "A*x-b" in the command line to check your constraint.

```
A*x-b=
    -72
      0
```

Often than not, a user writes a third file to define the constraints.

For Lab 1, use Helps → Demos → Optimization→ Tutorial for the Optimization Toolbox. One needs to go through all examples before solving the lab problems. For interested students, please refer the Optimization Toolbox User's Guide to learn more [1-2].

2.6 Optimization GUI

After you become familiar with the optimization process and program structures, you can enter "**optimtool**" from the command line to launch the optimization graphic user interface (GUI). This tool allows you to toggle between different options and observe the impact of these options on the optimization process and the optimal results.

Lab 2 Matlab Optimization Toolbox

1. After starting Matlab, use Helps → Demos → Optimization→ Tutorial for the Optimization Toolbox. One needs to go through all examples before solving the lab problems. Save all the example problems for random check.
2. Use the two functions "fminunc" and "fmincon" in Matlab to solve the following problems and report the results for 5 different runs for each problem with all the default parameter settings. Then experiment with different starting points. Results to report, along with your conditions, include the following:
   - obtained optimum
   - number of iterations, and
   - number of function evaluations.

   Problems:

1) $\text{Min } f = (x_1^2 + 12x_2 - 1)^2 + (49x_1^2 + 49x_2^2 + 84x_1 + 2324x_2 - 681)^2$

2) $\text{Min } f = (x_1 + 10x_2)^2 + 5(x_3 - x_4)^2 + (x_2 - 2x_3)^4 + 10(x_1 - x_4)^4$

3)

$$\text{Min } \ f = 4x_1^2 + x_1 - x_2 - 2.5$$
$$S.T. \ \ x_2^2 + 2x_1 - 1.5x_1^2 - 1 \geq 0 \quad \Rightarrow \quad -x_2^2 - 2x_1 + 1.5x_1^2 + 1 \leq 0$$
$$-x_2^2 - 2x_1^2 + 2x_1 + 4.25 \geq 0 \quad \Rightarrow \quad x_2^2 + 2x_1^2 - 2x_1 - 4.25 \leq 0$$

4) $\text{Min } \boldsymbol{f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2}$
   S.T. $\boldsymbol{x_1^2 + x_2^2 \leq 1}$

5) $\text{Min } \ f = -[25(x_1 - 2)^2 + (x_2 - 2)^2 + (x_3 - 1)^2 + (x_4 - 4)^2 + (x_5 - 1)^2 + (x_6 - 4)^2]$
   S.T.

$$x_{1,2} \geq 0, \quad 1 \leq x_3 \leq 5, \quad 0 \leq x_4 \leq 6, \quad 1 \leq x_5 \leq 5, \quad 0 \leq x_6 \leq 10, \quad 2 \leq x_1 + x_2 \leq 6, \quad x_1 - 3x_2 \leq 2,$$
$$4 \leq (x_3 - 3)^2 + x_4 \Rightarrow -(x_3 - 3)^2 - x_4 + 4 \leq 0$$
$$4 \leq (x_5 - 3)^2 + x_6 \Rightarrow -(x_5 - 3)^2 - x_6 + 4 \leq 0$$

$$\begin{cases} -x_1 - x_2 \leq -2 & (1) \\ x_1 + x_2 \leq 6 & (2) \end{cases} \qquad \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

3. For Problem 3 above, choose a "nominal condition," which is assumed to be all default settings (type optimoptions('fmincon') to see all the default options). Report five runs of results when changing each one of the following: 1) stopping criteria (OptimalityTolerance and MaxFunctionEvaluations), 2) approximate and exact derivatives (SpecifyObjectiveGradient), and 3) different Hessian methods (HessianApproximation). Comment on the results. (Hint: you could automate or semi-automate the repetitive runs and results reporting with Matlab programming.)

4. Write a report and submit it.

3)

$$\Rightarrow -n_2^2 - 2n_1 + 1.5n_1^2 + 1 \leq 0$$

$$\Rightarrow n_2^2 + 2n_1^2 - 2n_1 - 4.25 \leq 0$$

$An = B$

$3 \times 6 \quad \times 6 \times 1$

3 Global Optimization Toolbox -- Genetic Algorithm

Genetic algorithms (GAs) are stochastic global search and optimization methods that mimic the metaphor of natural biological evolution [3]. GAs work on a population of potential solutions and apply the principle of survival of the fittest to produce successively better approximations to a solution. At each step, a new population is created according to the level of fitness of the individuals in the problem domain. Over successive generations, the population "evolves" toward creating individuals that are better suited to their environment than those from which they were created. This chapter contains excerpts from references [1-2] as a shortened description. For details, one should refer to [1-2].

The genetic algorithm uses three main rules at each step to create the next generation from the current population:

- **Selection rules** select the individuals, called parents that contribute to the population at the next generation.
- **Crossover rules** combine two parents to form children for the next generation.
- **Mutation rules** apply random changes to individual parents to form children.

The GA toolbox in MATLAB is included in the Global Optimization toolbox. To use this toolbox, you can either create programs or use GA GUI. To differ a bit from the last chapter about the optimization toolbox, this time, we start directly from the GUI. To open the GA GUI you can type "**gatool**" in the command line, or you can find "ga - Genetic Algorithm" from the solver pull-down menu in "optimtool".

For using GA for optimization, you must first enter the following information:

Fitness function — The objective function you want to minimize. Enter the fitness function in the form @fitnessfun, where fitnessfun.m is an M-file that computes the fitness function. The @ sign creates a function handle to fitnessfun. The format of this function is no different from the objective function that you have created for the optimization toolbox. In fact, you can use one of the functions you created before as the fitness function here for a trial.

Number of variables — The length of the input vector to the fitness function.

You can enter constraints or a nonlinear constraint function for the problem in the Constraints pane. If the problem is unconstrained, leave these fields blank. To run the genetic algorithm, click the Start button. The tool displays the results of the optimization in the Run solver and view results pane.

3.1 Improving the results by tuning Options

You can change the options for the genetic algorithm in the Options pane. In fact, to get the best results from the genetic algorithm, you usually need to experiment with different options. Selecting the best options for a problem involves trial and error.

One of the most important factors that determine the performance of the genetic algorithm is the diversity of the population. If the average distance between individuals is large/small, the diversity is high/low. Getting the right amount of diversity is a matter of trial and error. If the diversity is too high or too low, the genetic algorithm might not perform well. The following will explain the options available in the GA toolbox and a few tips for determining the better value for them.

3.1.1 Population Options

Population type (PopulationType) specifies the data type of the input to the fitness function. You can set the Population type to be one of the following:

- Double Vector ('doubleVector') — Use this option if the individuals in the population have the type "double." This is the default.
- Bit string ('bitstring') — Use this option if the individuals in the population are bit strings.
- Custom ('custom') — Use this option to create a population whose data type is neither of the preceding.

You can also use a custom population type, in that case you must write your own creation, mutation, and crossover functions that accept inputs of that population type and specify these functions in the following fields, respectively:

Creation function (CreationFcn)
Mutation function (MutationFcn)
Crossover function (CrossoverFcn)

Population size (PopulationSize) specifies how many individuals there are in each generation. With a large population size, the genetic algorithm searches the solution space more thoroughly, thereby reducing the chance that the algorithm will return a local minimum that is not a global minimum. However, it will cause the algorithm to run more slowly.

Creation function (CreationFcn) specifies the function that creates the initial population for ga. You can choose from the following functions:

- Uniform (@gacreationuniform) creates a random initial population with a uniform distribution. This is the default if there are no constraints or bound constraints.
- Feasible population (@gacreationlinearfeasible) creates a random initial population that satisfies all bounds and linear constraints. It is biased to create individuals that are

on the boundaries of the constraints and to create well-dispersed populations. This is the default if there are linear constraints.
- Custom enables you to write your own creation function, which must generate data of the type that you specify in Population type.

Initial population (InitialPopulation) specifies an initial population for the genetic algorithm. The default value is [], in which case ga uses the default Creation function to create an initial population. If you enter a nonempty array in the Initial population field, the array must have no more rows than Population size and exactly the same number of columns as the number of variables. In this case, the genetic algorithm calls a Creation function to generate the remaining individuals, if required.

Initial scores (InitialScores) specifies initial scores for the initial population.

Initial range (PopInitRange) specifies the range of the vectors in the initial population that is generated by a creation function. You can set the Initial range to be a matrix with two rows and the number of columns to be the Number of variables. Each column has the form [lb; ub], where lb is the lower bound and ub is the upper bound for the entries in that coordinate.

3.1.2 Fitness Scaling Options

Fitness scaling converts the raw fitness scores that are returned by the fitness function to values in a range that is suitable for the selection function. The selection function uses the scaled fitness values to select the parents of the next generation. The selection function assigns a higher probability of selection to individuals with higher scaled values.

The range of the scaled values affects the performance of the genetic algorithm. If the scaled values vary too widely, the individuals with the highest scaled values reproduce rapidly, taking over the population gene pool too quickly and preventing the genetic algorithm from searching other areas of the solution space. On the other hand, if the scaled values vary only a little, all individuals have approximately the same chance of reproduction and the search will progress slowly.

Scaling function (FitnessScalingFcn) specifies the function that performs the scaling. The options are:
- Rank (@fitscalingrank) — The default fitness scaling function, Rank, scales the raw scores based on the rank of each individual instead of its score. The rank of an individual is its position in the sorted scores. The rank of the most fit individual is 1, the next most fit is 2, and so on. Rank fitness scaling removes the effect of the spread of the raw scores.

- Proportional (@fitscalingprop) — Proportional scaling makes the scaled value of an individual proportional to its raw fitness score.
- Top (@fitscalingtop) — Top scaling scales the top individuals equally. Selecting Top displays an additional field, Quantity, which specifies the number of individuals that are assigned positive scaled values. Quantity can be an integer between 1 and the population size or a fraction between 0 and 1 specifying a fraction of the population size. The default value is 0.4. Each of the individuals that produce offspring is assigned an equal scaled value, while the rest are assigned the value 0.
- Shift linear (@fitscalingshiftlinear) — Shift linear scaling scales the raw scores so that the expectation of the fittest individual is equal to a constant multiplied by the average score. You should specify the constant in the Max survival rate field, which is displayed when you select Shift linear.
- Custom enables you to write your own scaling function.

3.1.3 Selection Options

Selection options specify how the genetic algorithm chooses parents for the next generation. You can specify the function the algorithm uses in the Selection function (SelectionFcn) field in the Selection options pane. The options are:

- Stochastic uniform (@selectionstochunif) — The default selection function, Stochastic uniform, lays out a line in which each parent corresponds to a section of the line of length proportional to its scaled value. The algorithm moves along the line in steps of equal size. At each step, the algorithm allocates a parent from the section it lands on. The first step is a uniform random number less than the step size.
- Remainder (@selectionremainder) — Remainder selection assigns parents deterministically from the integer part of each individual's scaled value and then uses roulette selection on the remaining fractional part. For example, if the scaled value of an individual is 2.3, that individual is listed twice as a parent because the integer part is 2. After parents have been assigned according to the integer parts of the scaled values, the rest of the parents are chosen stochastically. The probability that a parent is chosen in this step is proportional to the fractional part of its scaled value.
- Uniform (@selectionuniform) — Uniform selection chooses parents using the expectations and number of parents. Uniform selection is useful for debugging and testing, but is not a very effective search strategy.
- Roulette (@selectionroulette) — Roulette selection chooses parents by simulating a roulette wheel, in which the area of the section of the wheel corresponding to an individual is proportional to the individual's expectation. The algorithm uses a random number to select one of the sections with a probability equal to its area.
- Tournament (@selectiontournament) — Tournament selection chooses each parent by choosing the number of players equal to the Tournament size at random and then choosing the best individual out of that set to be a parent. The tournament size must be at least two. The default value of Tournament size is four.

- Custom enables you to write your own selection function.

3.1.4 Reproduction Options

Reproduction options specify how the genetic algorithm creates children for the next generation.

Elite count (EliteCount) specifies the number of individuals that are guaranteed to survive to the next generation. Set Elite count to be a positive integer less than or equal to the population size. The default value is two.

Crossover fraction (CrossoverFraction) specifies the fraction of the next generation, other than elite children, that are produced by crossover. Set Crossover fraction to be a fraction between 0 and 1. The default value is 0.8.

3.1.5 Mutation Options

Mutation options specify how the genetic algorithm makes small random changes in the individuals in the population to create mutation children. Mutation provides genetic diversity and enables the genetic algorithm to search a broader space. You can specify the mutation function in the Mutation function (MutationFcn) field in the Mutation options pane. You can choose from the following functions:

- Gaussian (mutationgaussian) — The default mutation function, Gaussian, adds a random number taken from a Gaussian distribution with mean 0 to each entry of the parent vector. The standard deviation of this distribution is determined by the parameters Scale and Shrink, which are displayed when you select Gaussian, and by the Initial range setting in the Population options.

  The Scale parameter determines the standard deviation at the first generation. If you set Initial range to be a 2-by-1 vector v, the initial standard deviation is the same at all coordinates of the parent vector and is given by Scale*(v(2) - v(1)).

  If you set Initial range to be a vector v with two rows and the number of columns equal to the Number of variables, the initial standard deviation at coordinate i of the parent vector is given by Scale*(v(i,2) - v(i,1)).

  The Shrink parameter controls how the standard deviation shrinks as generations go by. If you set Initial range to be a 2-by-1 vector, the standard deviation at the *k*th generation, σ_k, is the same at all coordinates of the parent vector and is given by the recursive formula:

$$\sigma_k = \sigma_{k-1}\left(1 - Shrink\frac{k}{Generations}\right)$$

If you set Initial range to be a vector with two rows and the number of columns equal to the Number of variables, the standard deviation at coordinate *i* of the parent vector at the *k*th generation, $\sigma_{i,k}$, is given by the recursive formula:

$$\sigma_{i,k} = \sigma_{i,k-1}(1 - Shrink\frac{k}{Generations})$$

If you set Shrink to 1, the algorithm shrinks the standard deviation in each coordinate linearly until it reaches 0 at the last generation. A negative value of Shrink causes the standard deviation to grow.

- Uniform (mutationuniform) — Uniform mutation is a two-step process. First, the algorithm selects a fraction of the vector entries of an individual for mutation, where each entry has a probability Rate of being mutated. The default value of Rate is 0.01. In the second step, the algorithm replaces each selected entry by a random number selected uniformly from the range for that entry.
- Adaptive Feasible (mutationadaptfeasible) randomly generates directions that are adaptive with respect to the last successful or unsuccessful generation. The feasible region is bounded by the constraints and inequality constraints. A step length is chosen along each direction so that linear constraints and bounds are satisfied.
- Custom enables you to write your own mutation function.

3.1.6 Crossover Options

Crossover options specify how the genetic algorithm combines two individuals, or parents, to form a crossover child for the next generation.

Crossover function (CrossoverFcn) specifies the function that performs the crossover. You can choose from the following functions:

- Scattered (@crossoverscattered), the default crossover function, creates a random binary vector and selects the genes where the vector is a 1 from the first parent, and the genes where the vector is a 0 from the second parent and combines the genes to form the child. For example, if p_1 and p_2 are the parents:

  p1 = [a b c d e f g h]
  p2 = [1 2 3 4 5 6 7 8]

  and the binary vector is [1 1 0 0 1 0 0 0], the function returns the following child:

  child1 = [a b 3 4 e 6 7 8]

- Single point (@crossoversinglepoint) chooses a random integer *n* between 1 and Number of variables and then:
  - Selects vector entries numbered less than or equal to *n* from the first parent.
  - Selects vector entries numbered greater than *n* from the second parent.
  - Concatenates these entries to form a child vector.

For example, if p_1 and p_2 are the parents:

p1 = [a b c d e f g h]
p2 = [1 2 3 4 5 6 7 8]

and the crossover point is 3, the function returns the following child:

child = [a b c 4 5 6 7 8]

- Two point (@crossovertwopoint) selects two random integers *m* and *n* between 1 and Number of variables. The function selects:
    - Vector entries numbered less than or equal to *m* from the first parent
    - Vector entries numbered from *m*+1 to *n*, inclusive, from the second parent
    - Vector entries numbered greater than n from the first parent.

The algorithm then concatenates these genes to form a single gene. For example, if p_1 and p_2 are the parents:

p1 = [a b c d e f g h]
p2 = [1 2 3 4 5 6 7 8]

and the crossover points are 3 and 6, the function returns the following child:

child = [a b c 4 5 6 g h]

- Intermediate (@crossoverintermediate) creates children by taking a weighted average of the parents. You can specify the weights by a single parameter, **Ratio**, which can be a scalar or a row vector of length equal to Number of variables. The default is a vector of all 1's. The function creates the child from parent1 and parent2 using the following formula:

child = parent1 + rand * Ratio * (parent2 - parent1)

If all the entries of Ratio lie in the range [0, 1], the produced children are within the hypercube defined by placing the parents at opposite vertices. If Ratio is not in that range, the children might lie outside the hypercube. If Ratio is a scalar, then all the children lie on the line between the parents.

- Heuristic (@crossoverheuristic) returns a child that lies on the line containing the two parents, a small distance away from the parent with the better fitness value in the

direction away from the parent with the worse fitness value. You can specify how far the child is from the better parent by the parameter Ratio, which appears when you select Heuristic. The default value of Ratio is 1.2. If parent1 and parent2 are the parents, and parent1 has the better fitness value, the function returns the child:

child = parent2 + R * (parent1 - parent2);

- Arithmetic (@crossoverarithmetic) creates children that are the weighted arithmetic mean of two parents. Children are always feasible with respect to linear constraints and bounds.
- Custom enables you to write your own crossover function.

3.1.7 Migration Options

Migration options specify how individuals move between subpopulations. Migration occurs if you set Population size to be a vector of length greater than 1. When migration occurs, the best individuals from one subpopulation replace the worst individuals in another subpopulation. Individuals that migrate from one subpopulation to another are copied. They are not removed from the source subpopulation.

You can control how migration occurs by the following three fields in the Migration options pane:

Direction (MigrationDirection) — Migration can take place in one or both directions.
If you set Direction to Forward ('forward'), migration takes place toward the last subpopulation. That is, the *n*th subpopulation migrates into the (*n*+1)th subpopulation.

If you set Direction to Both ('both'), the *n*th subpopulation migrates into both the (*n*–1)th and the (*n*+1)th subpopulation.

Migration wraps at the ends of the subpopulations. That is, the last subpopulation migrates into the first, and the first may migrate into the last.

Interval (MigrationInterval) — Specifies how many generations pass between migrations. For example, if you set Interval to 20, migration takes place every 20 generations.
Fraction (MigrationFraction) — Specifies how many individuals move between subpopulations. Fraction specifies the fraction of the smaller of the two subpopulations that moves. For example, if individuals migrate from a subpopulation of 50 individuals into a subpopulation of 100 individuals and you set Fraction to 0.1, the number of individuals that migrate is 0.1 * 50 = 5.

3.1.8 Algorithm Settings

Algorithm settings define algorithmic specific parameters. Parameters that can be specified for a nonlinear constraint algorithm include:

Initial penalty (InitialPenalty) — Specifies an initial value of the penalty parameter that is used by the algorithm. Initial penalty must be greater than or equal to 1.
Penalty factor (PenaltyFactor) — Increases the penalty parameter when the problem is not solved to the required accuracy and constraints are not satisfied. The Penalty factor must be greater than 1.

3.1.9 Hybrid Function Options

A hybrid function is another minimization function that runs after the genetic algorithm terminates. You can specify a hybrid function in Hybrid function (HybridFcn) options. The choices are:

- [] — No hybrid function.
- fminsearch (@fminsearch) — Uses the MATLAB® function fminsearch to perform unconstrained minimization.
- patternsearch (@patternsearch) — Uses a pattern search to perform constrained or unconstrained minimization.
- fminunc (@fminunc) — Uses the Optimization Toolbox function fminunc to perform unconstrained minimization.
- fmincon (@fmincon) — Uses the Optimization Toolbox function fmincon to perform constrained minimization.

3.1.10 Introduction to Multiobjective Optimization

When dealing with multiple objectives, one will enter a situation that there are multiple or numerous solutions. For example, for the problem below we plot the function using the graphic method as shown in Figure 3-1.

Minimize: $F(x) = \{f_1(x), f_2(x)\}$

$$f_1(x_1, x_2) = (x_1 + x_2 - 7.5)^2 + (x_2 - x_1 + 3)^2 / 4$$

$$f_2(x_1, x_2) = (x_1 - 1)^2 / 4 + (x_2 - 4)^2 / 2$$

Subject to:

$$g_1(x_1, x_2) = 2.5 - (x_1 - 2)^3 / 2 - x_2 \geq 0$$

$$g_2(x_1, x_2) = 3.85 + 8(x_2 - x_1 + 0.65)^2 - x_2 - x_1 \geq 0$$

$$0 \leq x_1 \leq 5 \qquad 0 \leq x_2 \leq 3$$

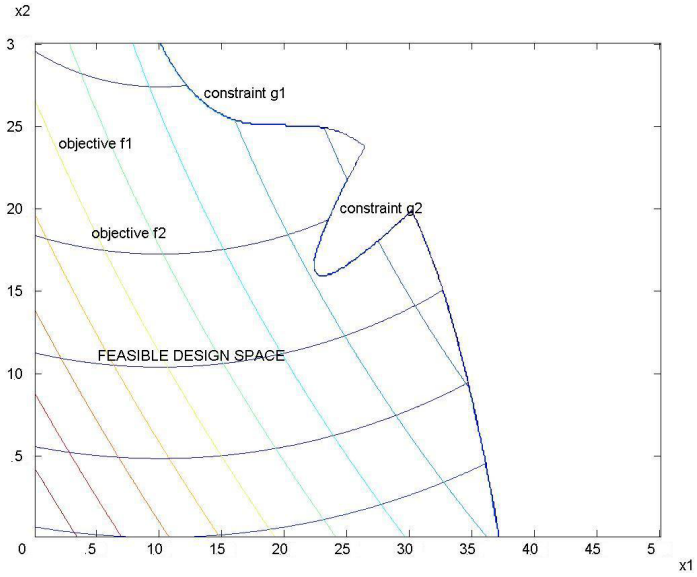

Figure 3-1 Design space of the example problem.

This figure shows two sets of contour lines for the two objective fucntions, respectively, as well as the two constraints. If we map the feasible design space to the function space (or performance space), we have the following figure.

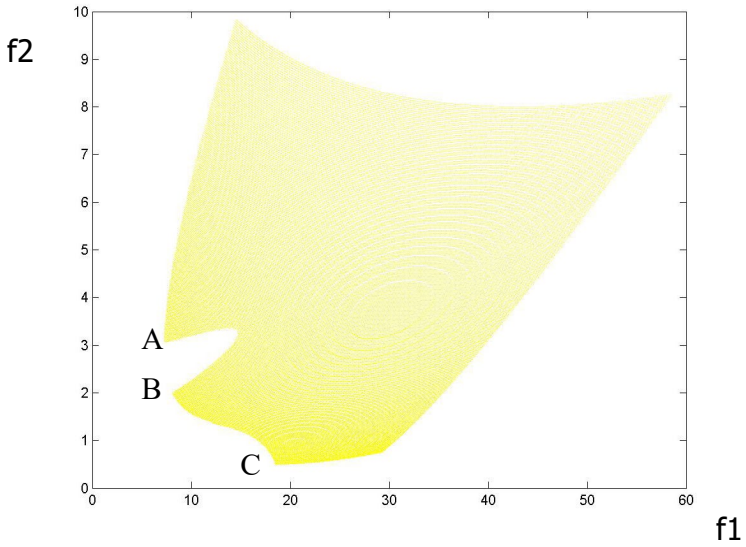

Figure 3-2 Performance space of the example problem.

The yellow area corresponds to the feasible design space in Figure 3-1. As one can see when one improves f1, f2 may be worse and vice versa. In the end, the optimal solution for a multiobjective optimization problem is a Pareto set, or a curve for two objectives, a surface for three objectives, and so on. In Figure 3-2, the curve segment BC and the point A constitute the Pareto set, or often referred as Pareto front/frontier. This means any other

point in the yellow space will have inferior performances (or objective function values) than at least one solution in the Pareto set.

This could be easily understood with a real-life example. Assume that a car manufacturer wants to optimize both the performance and cost. On one extreme, one can make a high-performance high cost car (e.g. Ford Mustang), and the other extreme a low-performance but low-cost car (e.g., Toyota Yaris). Both are optimal considering these two objectives simulateneously. Then there are many falling between the two extremes such as Honda Civic, Toyota Camry, etc. They could all be a point in the Pareto set.

3.1.11 Multiobjective optimization options

Multiobjective options define parameters characteristic of the multiobjective genetic algorithm. You can specify the following parameters:

DistanceMeasureFcn — Defines a handle to the function that computes distance measure of individuals, computed in decision variable, design space (genotype), or in function space (phenotype). For example, the default distance measure function is distancecrowding in function space, or {@distancecrowding,'phenotype'}. This distance measures the distance between two neighbouring points in the Pareto frontier.

ParetoFraction — Sets the fraction of individuals to keep on the first Pareto frontier while the solver selects individuals from higher frontiers. This option is a scalar between 0 and 1.

3.1.12 Stopping Criteria Options

Stopping criteria determine what causes the algorithm to terminate. You can specify the following options:

Generations (Generations) — Specifies the maximum number of iterations for the genetic algorithm to perform. The default is 100.

Time limit (TimeLimit) — Specifies the maximum time in seconds the genetic algorithm runs before stopping.

Fitness limit (FitnessLimit) — The algorithm stops if the best fitness value is less than or equal to the value of Fitness limit.

Stall generations (StallGenLimit) — The algorithm stops if the weighted average change in the fitness function value over Stall generations is less than Function tolerance.

Stall time limit (StallTimeLimit) — The algorithm stops if there is no improvement in the best fitness value for an interval of time in seconds specified by Stall time.

Function tolerance (TolFun) — The algorithm runs until the cumulative change in the fitness function value over Stall generations is less than or equal to Function Tolerance.

Nonlinear constraint tolerance (TolCon) — The Nonlinear constraint tolerance is not used as stopping criterion. It is used to determine the feasibility with respect to nonlinear constraints.

3.2 Example

Suppose you want to minimize the simple fitness function of two variables x_1 and x_2,

$$\min f(x) = 100\ (x_1^2 - x_2^2)^2 + (1 - x_1)^2$$

subject to the following nonlinear inequality constraints and bounds:

$$x_1x_2+x_1-x_2+1.5\leq 0$$
$$10-x_1x_2\leq 0$$
$$0\leq x_1\leq 1$$
$$0\leq x_2\leq 13$$

Begin by creating the fitness and constraint functions. First, create an M-file named simple_fitness.m as follows:

```
function y = simple_fitness(x)
y = 100*(x(1)^2 - x(2))^2 + (1 - x(1))^2;
```

The genetic algorithm function, `ga`, assumes the fitness function will take one input x, where x has as many elements as the number of variables in the problem. The fitness function computes the value of the function and returns that scalar value in its one return argument, y.

Then create an M-file, simple_constraint.m, containing the constraints:

```
function [c, ceq] = simple_constraint(x)
c = [1.5 + x(1)*x(2) + x(1) - x(2);...
-x(1)*x(2) + 10];
ceq = [];
```

The `ga` function assumes the constraint function will take one input `x`, where x has as many elements as the number of variables in the problem. The constraint function computes the values of all the inequality and equality constraints and returns two vectors, `c` and `ceq`, respectively.

To minimize the fitness function, you need to pass a function handle to the fitness function as the first argument to the `ga` function and specify the number of variables as the second argument. Lower and upper bounds are provided as `LB` and `UB`, respectively. In addition, you also need to pass a function handle to the nonlinear constraint function.

```
ObjectiveFunction = @simple_fitness;
```

```
nvars = 2;    % Number of variables
LB = [0 0];   % Lower bound
UB = [1 13];  % Upper bound
ConstraintFunction = @simple_constraint;
[x,fval]=
ga(ObjectiveFunction,nvars,[],[],[],[],LB,UB,ConstraintFunction)

Warning: 'mutationgaussian' mutation function
is for unconstrained minimization only;
using @mutationadaptfeasible mutation function.
Set @mutationadaptfeasible as MutationFcn options
using GAOPTIMSET.

Optimization terminated: current tolerance on f(x) 1e-007
is less than options.TolFun and constraint violation is
less than options.TolCon.

x =
    0.8122   12.3122

fval =
  1.3578e+004
```

3.3 A quick reference

Below is a quick reference for global optimization for single and multiple objective functions using GA.

For problems with real variables (non-integers):
X=ga(fitnessfcn,nvars,A,b,Aeq,beq,LB,UB,nonlcon,options)
Ax<=b
Aeq*x=beq

For problems with integer variables:
X=ga(fitnessfcn,nvars,A,b,[],[],LB,UB,nonlcon,IntCon,options)
Note: when you have integer variables: no linear or nonlinear equality constraints is allowed.

For problems with multiple objective functions:
X=gamultiobj(fitnessfcn,nvars,A,b,Aeq,beq,LB,UB,nonlcon,options)

Changing the options:
Options=gaoptimset('param1',value1,'param2',value2,…)

1. Population size:
   Options=gaoptimset('PopulationSize',100);

2. Mutation rate (fraction):
   Options=gaoptimset('MutationFraction',{@mutationuniform,0.01});
   The value not too low: 0.0000001
   The value not too high: 0.2,0.4,0.5
   Recommended value range: 0.01- 0.1
3. Crossover rate (fraction):
   Options=gaoptimset('CrossoverFraction',0.8);
   Value range: 0-1 (Recommended: 0.6)

   To see the default values, simply type the following in the command line:
   gaoptimset

Lab 3 Matlab Genetic Algorithm Toolbox

1. Read the Help → Content → Genetic Algorithms and Direct Search → Using the Genetic Algorithm, run the examples, and save them for random check.
2. Use the GA toolbox in Matlab to solve the following problems and report the following:
   - Obtained global optimum
   - Number of iterations
   - Number of function evaluations, and
   - Your options

1) Problems of discrete variables: Compound gear train (GT) design problem

This is a discrete variant problem involving a compound gear train design as shown in the following figure.

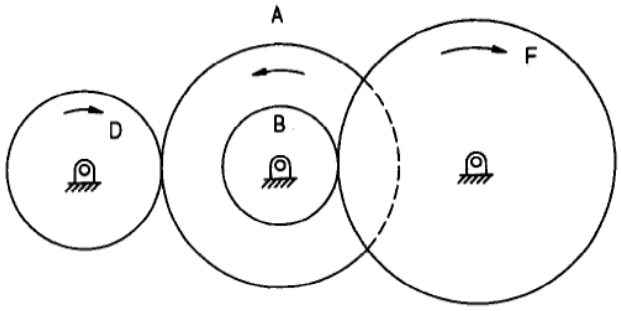

Figure 3-3 Compound gear train

It is desired to produce a gear ratio as close as possible to 1/6.931. For each gear, the number of teeth must be an integer between 14 and 60. The integer design variables are the numbers of teeth, $x=[T_d, T_b, T_a, T_f]^T=[x_1, x_2, x_3, x_4]^T$. The optimization problem is formulated as:

$$\text{Minimize} \quad GT(X)=\left(\frac{1}{6.931}-\frac{x_1x_2}{x_3x_4}\right)^2$$

Subject to

$$14 \le x_i \le 60, \quad i=1,2,3,4$$

2) A problem of high dimension: a function of 16 variables with x_i in [-1 1]

$$f_{F16}(x)=\sum_{i=1}^{16}\sum_{j=1}^{16}a_{ij}(x_i^2+x_i+1)(x_j^2+x_j+1),\ i,j=1,\ 2,\ \ldots,\ 16,$$

what is the process?

$$[a_{ij}]_{\text{row 1–8}} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix},$$

$$[a_{ij}]_{\text{row 9–16}} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

3) Multi-objective function without constraints:

$$\text{Min} \quad f_1 = 1 - \exp\left(-\sum_{i=1}^{3}\left(x_i - \frac{1}{\sqrt{3}}\right)^2\right)$$

$$f_2 = 1 - \exp\left(-\sum_{i=1}^{3}\left(x_i + \frac{1}{\sqrt{3}}\right)^2\right)$$

4) Multi-objective function with constraints

Min

$$f_1(x) = 1 - e^{-4x_1}\sin^6(6\pi x_1)$$

$$f_2(x) = \left(\frac{f_1(x)}{g(x)}\right)^2$$

$$g(x) = 1 + 9\left[\sum_{i=2}^{n} x_i \Big/ (n-1)\right]^{0.25}$$

$$0 \le x_i \le 1, i = 1, \ldots, n = 10$$

3. For Problem 2, try different population size, mutation rate, crossover rate, and report your findings.

4. Write a report on the results, analysis, and findings.

4 OASIS

OASIS (Optimization Assisted System Integration Software) is developed for computationally intensive problems by Empower Operations Corp.

OASIS helps engineers find optimal product solutions based on given design variables and objectives. It is driven by the following key features:

- **Black-Box Objective Support:** OASIS can optimize systems even if no mathematically defined input-output relationship is known. OASIS intelligently uses simulations to model the system response mathematically while collecting sample-data (meta-modelling). This makes OASIS compatible with a wide variety of real world scenarios. Simply provide a link to a complex external simulation tool (e.g. ANSYS, MATLAB etc.), and OASIS will decipher the results and automatically suggest optimal values for your design variables.

- **Economic Use of Simulations:** OASIS's algorithms are especially tailored to find a global optimum with a ***minimal*** number of objective function evaluations. The state-of-the-art optimization algorithms iteratively phase out designs with poor performance early-on, ensuring that time is spent computing top-performing designs. OASIS can handle large-scale design problems with the number of variables up to a few hundrend, representing the state-of-the-art.

- **One-button Optimization:** OASIS automatically knows which algorithm and parameters to match with your optimization problem. Simply define your variables, objectives, and constraints, and OASIS will recognize if the problem is single or multi-objective. A user does not need to pick an algorithm or tune algorithm parameters such as population size, rates, coefficients, etc.

- **Multivariate Visualization:** OASIS provides insight about your problem and optimization in the Visulization window. The interactive parallel coordinate graphs provide a visual summary of the data collected during optimization, illustrating the effect of variables on your output. Furthermore, convergence information gives feedback of the optimization's progress as it executes.

Please refer to https://www.youtube.com/channel/UCXjlBAR7ZltPelBtYKJLh4Q for quick learning. Instruction of OASIS can be found via HELP in OASIS. Contact info@empowerops.com or visit www.empoweroperations.com for more information.

Lab 4 Application of OASIS in Simulation-based Design

This lab is composed of two parts. Part I involves applying several optimizers on two multi-dimensional, multi-modal optimization problems. Part II introduces you to optimization on simulations.

Part I: Black-Box Global Optimization

In this part you will apply several optimizers on black-box, multi-dimensional, and multi-modal optimization problems. A multi-dimensional problem has multiple variables, and a multi-modal problem has multiple local and/or global minima. A multi-modal problem has local minima that would trap a local optimizer, while global optimizers can escape local optimum regions. A black-box problem does not provide the optimizer gradient information except for the fitness values.

This class of problems poses significant challenges to deterministic gradient-based local optimization methods. First, algorithms such as BFGS and other non-linear programming methods rely on gradient information to efficiently converge to the optimum. When gradient information is not available, finite-differencing is applied to calculate gradients, which reduces the efficiency of algorithms. When the function output is noisy, such approximation of gradients will be errorneous and unreliable. Second, traditional deterministic gradient-based algorithms are largely local optimization methods. They converge to the same optimum given the same initial starting point. While there are workarounds to this problem, e.g., randomizing the starting point for multiple runs of the optimizer, these algorithms are intrinsically not global optimization schemes.

Stochastic global optimizers are typically used for solving black-box global optimization problems. Stochastic algorithms incorporate randomness into the optimization process and do not typically rely on gradient information. The Genetic Algorithm (GA) is an example of a stochastic global optimizer.

Optimizers

The optimizers used for this part are: MATLAB's fmincon(.), MATLAB's GA, and Empower Operations' SOGO (Single Objective Global Optimizer). Fmincon(.) is a local optimizer, while GA and SOGO are global optimizers. Ga(.) and fmincon(.) can be invoked through MATLAB. For instructions on how to use them, type the command: *help ga*, or *help fmincon*. The table below shows the recommended GA settings.

Table 1: GA Settings

Crossover Fraction	Mutation Fraction	Population Size	Stall Generation Limit
0.6	0.05	10	99

SOGO is part of the OASIS software package, the flagship product from Empower Operations Corp. OASIS is already installed on your computer and can be found in the start menu.

Optimization Problems

This lab includes two multi-dimensional and multi-modal optimization problems, which are treated as black-box functions for the purpose of comparison. All the optimizers treat them as black-box functions with fmincon(.) using the finite difference method to calculate the gradients and ga(.) and SOGO only using the function values.

Recall that you have coded the following two objective functions in Lab 1.

Keane's Bump Function

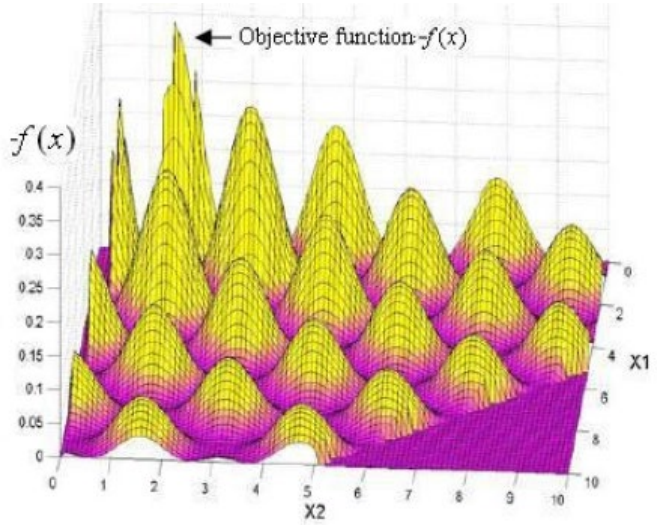

Figure 4-1: 2D Keane's Bump Function [1]

$$f(x) = -\left|\left[\sum_{i=1}^{d}\cos^4(x_i) - 2\prod_{i=1}^{d}\cos^2(x_i)\right] \Big/ \left(\sum_{i=1}^{d} i x_i^2\right)^{0.5}\right|$$

s.t.

$$0.75 - \prod_{i=1}^{d} x_i < 0$$

$$\sum_{i=1}^{d} x_i - 7.5d < 0$$

$$x \in [0, 10]; d = \#\, variables = 5$$

No known analytical optimum exists for this function.

Shubert Function

This function was modified to accommodate more than 2 dimensions.

$$f(\boldsymbol{x}) = \prod_{i=1}^{d} \sum_{j=1}^{5} j \cos\big((j+1)x_i + j\big)$$

$$x \in [-5.12, 5.12]; d = \#\, variables = 10$$

No known analytical optimum exists for this function.

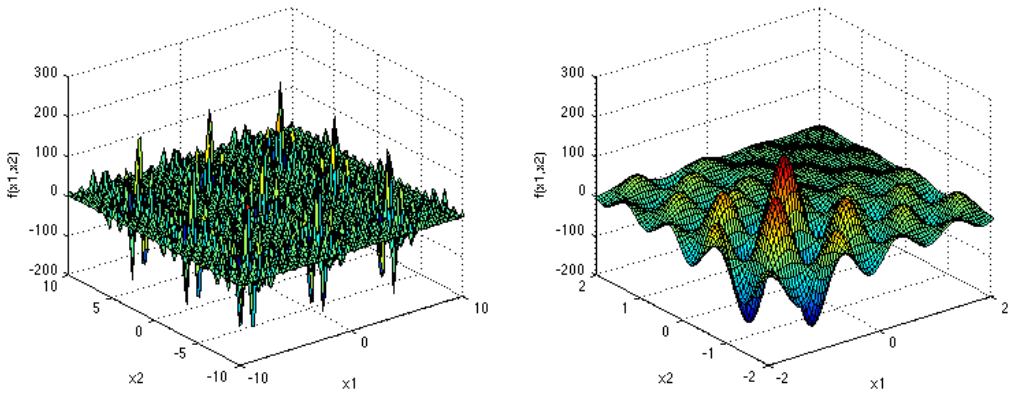

Figure 4-2: 2D Shubert Function [2]

Procedure

This is the step-by-step procedure to complete Part I:

1. Open MATLAB

2. Use the two functions that you have created in Lab 1 and write a script file that runs fmincon(.) and ga(.) of the above optimization problems. Perform two runs for each problem. Limit fmincon(.) and ga(.) to 1000 function evaluations. For fmincon(.), choose a random starting point for each run.

3. Save the resultant data into an Excel file, properly labelled.

4. Close MATLAB.

5. Open OASIS.

6. Setup an optimization flow using the Bump function (refer to the OASIS manual posted in CANVAS for detailed instructions). Set the optimization settings to do two runs with 1000 as the function evaluation limit.
7. Run the optimization.
8. Export optimization results to Excel (see OASIS manual).
9. Repeat steps 7-9 for the Shubert function (do not start a new project, use the existing project).
10. Save your project to a file.
11. Compare the results in the report.

Part II: Simulation-Based Optimization

In practice, many optimization problems do not have mathematical formulations. For example, an engineer that wants to optimize the dimensions of a critical bracket to meet stress requirements will most likely not be working with mathematical equations. Modern simulation software (e.g. ANSYS or COMSOL) is generally used to model the physics, and the simulation model is inherently black-box. The simulation model may also be computationally expensive so that each run of the simulation model can take anywhere from a fraction of a minute to many hours. In this practical scenario, a software package that can easily integrate with black-box simulations that also has an efficient optimizer is needed.

Part II of the workshop introduces you to a robot link design problem. You will be required to optimize the problem using OASIS.

Safety, deformation, and mass are of critical concerns when designing a robotic link. On the one hand, you need to have the most rigid body with less deformation which usually means more mass; on the other hand, the safety factor and minimum weight are of great importance. To reduce the mass, dimensions and fillets should be small while still satisfying safety and deformation constraints.

Figure 4-3 shows a robot link that has fixed supports and the plane that the forces and moments act on.

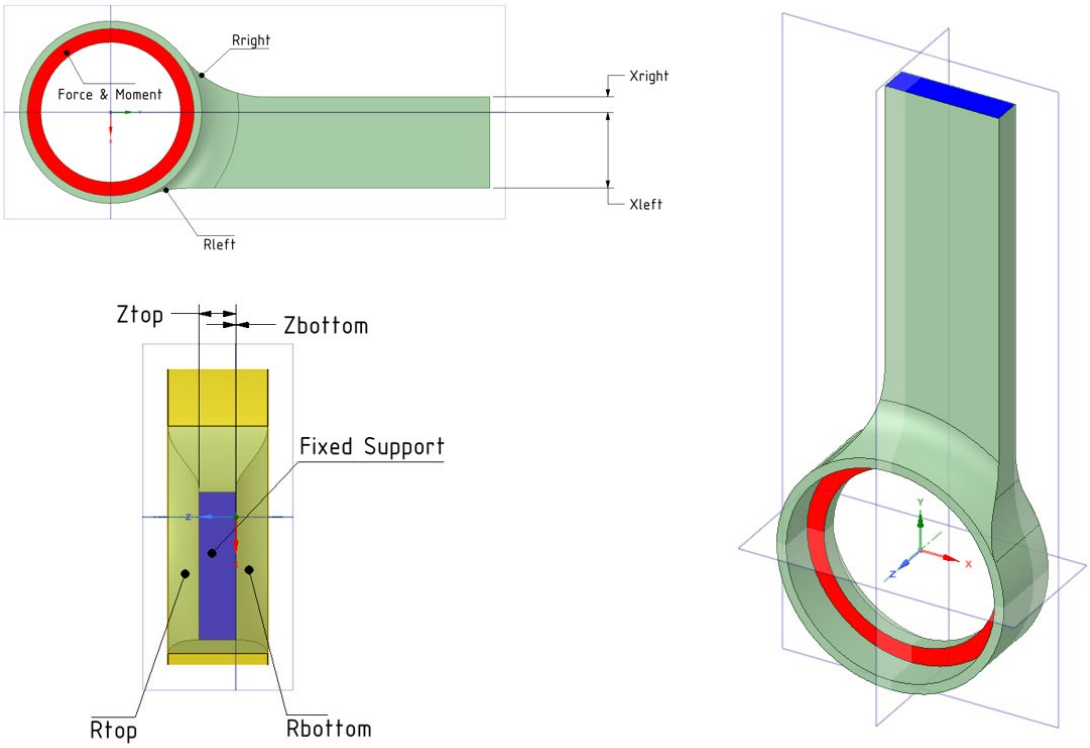

Figure 4-3 Simplified link with parameters and boundaries.

The variables are the Xright, Xleft (length) of block edges from the YOZ plane and the Ztop, Zbottom(height) distances from the XOY plane. Moreover, there are four additional variables for the fillet radii, Rright, Rleft, Rtop, and Rbottom.

The range of Xleft and Xright is set as [-59 59] mm; the range for Ztop and Zbottom is [-12 27] mm, and fillets' range is [1 80] mm.

ANSYS Static Structural module from ANSYS Workbench is the solver. The material is Aluminum Alloy. Force and Momentum vectors act on the red face while the link is fixed to the ground on the blue face.

The optimization model can be described as below:

Min *Link Mass*

Xleft, Xright, Ztop,Zbottom,Rleft,Rright, Rtop, Rbottom

Subject to

2.0 <= Safety Factor

Max Deformation <= 1 mm

20.0 mm <= Xleft – Xright

7.0 mm <= Ztop – Zbottom

Upper and Lower limits as described above

The optimization process can be described by the block diagram in Fig. 4-4.

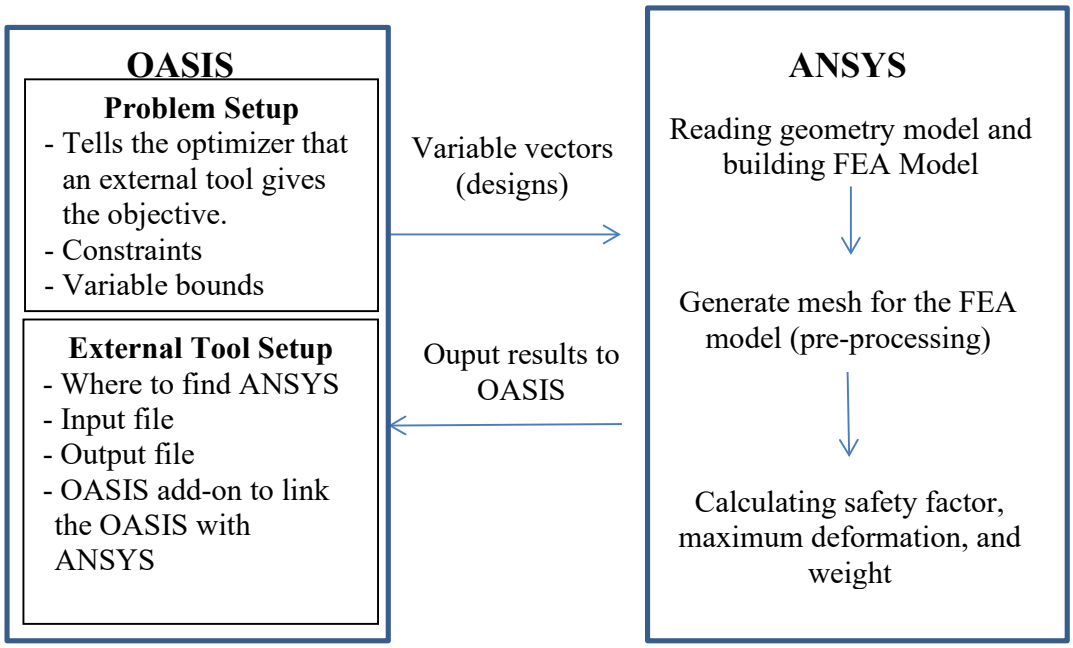

Figure 4-4 The optimization setup and interaction with FEA.

Procedure

This is the step-by-step procedure to complete Part II:

1. Understand the process of integrating an external tool within OASIS.
2. Download the lab files from CANVAS
3. Unzip the provided CAD and set up Model in ANSYS 19.1
4. Run OASIS add-on from ANSYS
5. Set up the optimization problem according to OASIS manual.
6. Run the optimization and report the objective function values when the number of function evaluations reaches 10, 30, 70, and 100.
7. Run the optimization with ANSYS Design of Experiment module
8. Compare the results and optimization time
9. Save the project file.

Lab Report and Result Submission

You are required to produce a lab report in a group of two, which is to be sent to the TA along with all of the MATLAB files, the OASIS project file (.oasis), and your Excel result file. Put all the required documents in a zip file. Name the zip file “last name1_last name 2_Lab 4”. Send the zip file to the TA or upload it to Canvas.

5 References

1. *Optimization Toolbox™ 5 User's Guide*, The MathWorks, Inc., Sept., 2010.
2. *Global Optimization Toolbox 3 User's Guide*, The MathWorks, Inc., 2010.
3. Holland, J., *Adaptation in Natural and Artificial Systems*, The University of Michigan Press, Ann Arbor, 1975.
4. Wang, L., Shan, S., and Wang, G. G., "Mode-Pursuing Sampling Method for Global Optimization on Expensive Black-box Functions," *Journal of Engineering Optimization*, Vol. 36, No. 4, August 2004, pp. 419-438.
5. Shan, S., and Wang, G. G., "An Efficient Pareto Set Identification Approach for Multi-objective Optimization on Black-box Functions," *ASME Transactions, Journal of Mechanical Design,* Vol. 127, No. 5, pp. 866-874, 2005.

Appendix: Report Format

Cover page

Your name, student no. and signature

Introduction (one paragraph)

Briefly mention the tasks of the lab and your approach.

Results and analysis

Provide major result comparisons, e.g., average results from all runs etc.

Conclusion (one paragraph)

Major technical findings and summary of what you have learned in the lab

References (optional)

Appendices

Provide detailed results for each run for at least one problem, if it tends to be lengthy.

Provide all of your codes developed for each problem.

Note:

1. All reports will be marked according to its technical correctness, quality, and completeness 80%.
2. Formatting and writing of the report will be marked out of 20%.

Common Mistakes to Avoid in Technical Writing

- Missing page number
- No reference; though there is reference, no citation of the references.
- Inconsistent types and sizes of fonts for the same type of content
- Inconsistent line spacing
- Typos, grammatical problems
- Missing captions for figures or tables