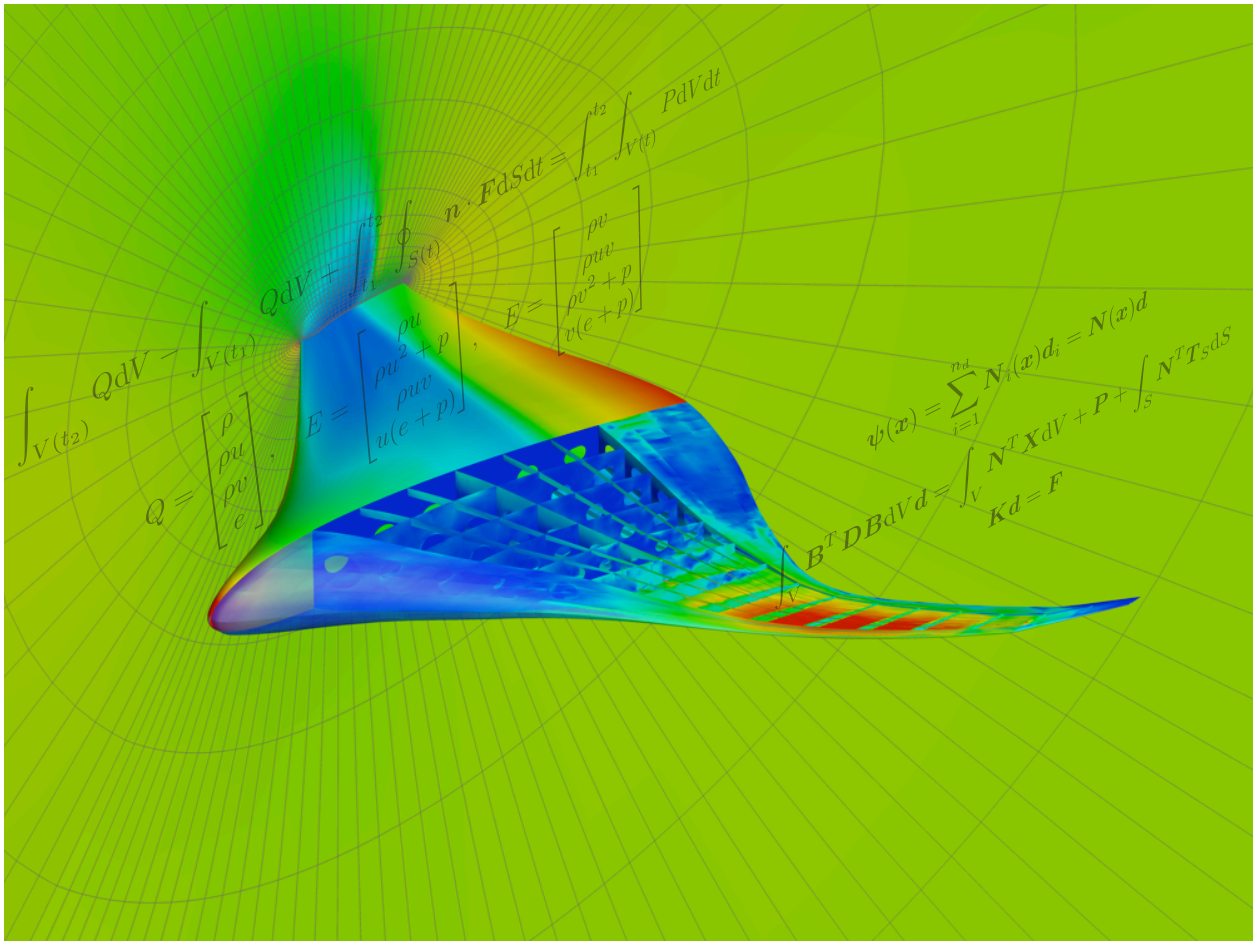
Multidisciplinary Design Optimization



Joaquim R. R. A. Martins

Andrew Ning, and Jason Hicken

Compiled on Thursday 5th January, 2017 at 20:58

Contents

Chapter 1

Introduction

1.1 What is "MDO"?

Multidisciplinary design optimization (MDO) is the application of numerical optimization techniques to the design of engineering systems that involve multiple disciplines. Aircraft are prime examples of multidisciplinary systems, so it is no coincidence that MDO emerged within the aerospace community.

Before covering MDO, we first need to cover the "O", i.e., numerical optimization. This consists in the use of algorithms to minimize or maximize a given function by varying a number of variables. The problem might or might not be subject to constraints.

Since many engineering design problems seek to maximize some measure of performance, it became obvious that using numerical optimization in design could be extremely useful, hence *design optimization* (the "DO" in MDO). Structural design was one of the first applications. A typical structural design optimization problem is to minimize the weight by varying structural thicknesses subject to stress constraints.

MDO emerged in the 1980s following the success of the application of numerical optimization techniques to structural design in the 1970s. Aircraft design was one of the first applications of MDO because there is much to be gained by the simultaneous consideration of the various disciplines involved (structures, aerodynamics, propulsion, stability and controls, etc.), which are tightly coupled. As an example, suppose that you are able to save one unit of weight in the structure. Because the coupled nature of all the aircraft weight dependencies, and the reduction in induced drag, the total reduction in the aircraft gross weight will be several times the structural weight reduction (about 5 for a typical airliner).

In spite of its usefulness, DO and MDO remain underused in industry. There are several reasons for this, one of which is the absence of MDO in undergraduate and graduate curricula. This is changing, as most top aerospace and mechanical engineering departments nowadays include at least one graduate level course on MDO. We have also seen MDO being increasingly used by the major airframers.

The design optimization process follows a similar iterative procedure to that of the conventional design process, with a few key differences. Fig. 1.1 illustrates this comparison. Both approaches must start with a baseline design derived from the specifications. This baseline design usually requires some engineering intuition and represents an initial idea. In the conventional design process this baseline design is analyzed in some way to determine its performance. This could involve numerical modeling or actual building and testing. The design is then evaluated based on the results and the designer then decides whether the design is good enough or not. If the answer is

no — which is likely to be the case for at least the first few iterations — the designer will change the design based on intuition, experience, or trade studies. When the design is satisfactory, the designer will arrive at the final design.

For more complex engineering systems, there are multiple levels and thus cycles in the design process. In aircraft design, these would correspond to the preliminary, conceptual, and detailed design stages.

The design optimization process can be pictured using the same flow chart, with modifications to some of the blocks. Instead of having the option to build a prototype, the analysis step must be completely numerical and must not involve any input from the designer. The evaluation of the design is strictly based on numerical values for the objective to be minimized and the constraints that need to be satisfied. When a rigorous optimization algorithm is used, the decision to finalize the design is made only when the current design satisfies the necessary optimality conditions that ensure that no other design "close by" is better. The changes in the design are made automatically by the optimization algorithm and do not require the intervention of the designer. On the other hand, the designer must decide in advance which parameters can be changed. In the design optimization process, it is crucial that the designer formulate the optimization problem well.

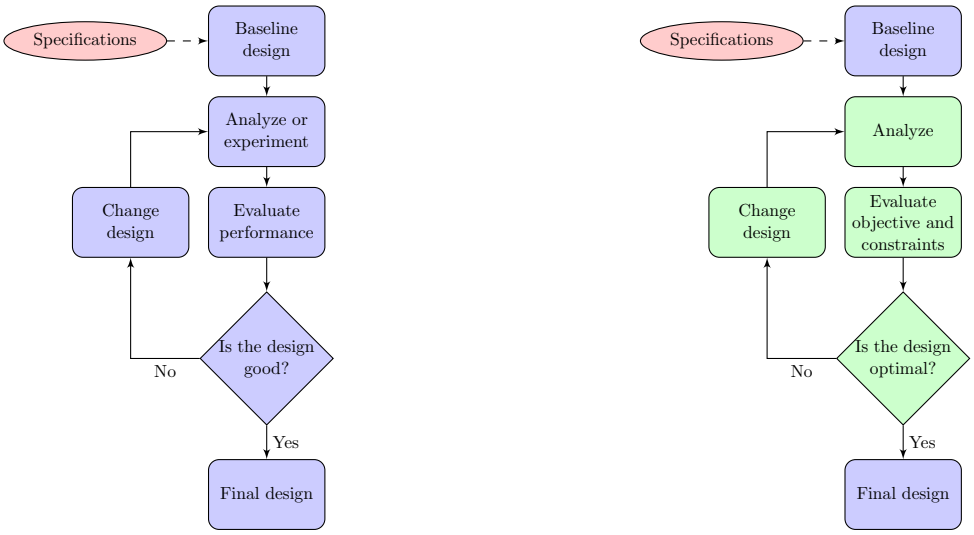


Figure 1.1: Conventional (left) versus optimal (right) design process

In this text we will generally frame problems and discussions in the context of engineering design. However, you should keep in mind that the optimization methods are quite general and are used in other applications that may not be typically classified as design problems (e.g., controls, machine learning, regression, etc.). In other words, we mean 'design' in a general sense where parameters are allowed to change towards an improving objective. We will now discuss the components of this formulation in more detail: the objective function, the constraints, and the design variables.

1.2 Terminology and Problem Statement

The formulation of the problem statement is often undervalued. Even experienced designers are sometimes not clear on what the different components in the optimization problem statement mean. Common conceptual errors include confusing constraints with objective functions, or selecting a misleading objective function. If we get the problem formulation wrong, then the solution of that problem could fail or simply return a mathematical optimum that is not the engineering optimum.

The "Disciplanes"
Is there *one* aircraft which is the fastest, most efficient, quietest, most inexpensive?

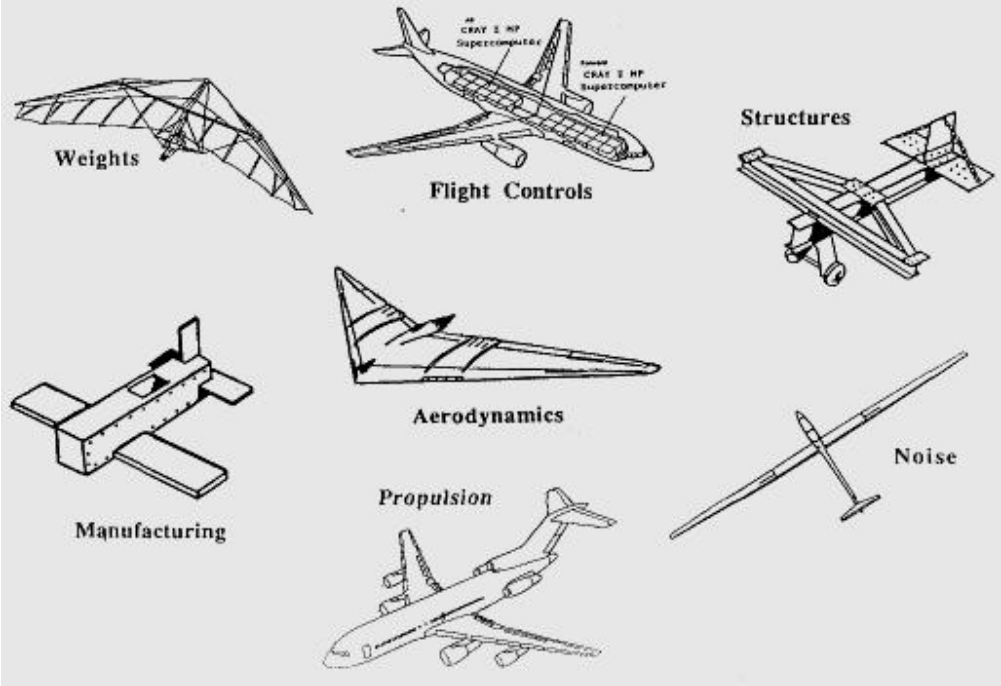


Figure 1.2: "You can only make one thing best at a time."

1.2.1 Objective Function

The objective function, f, is the scalar that we want to minimize. This is the convention in numerical optimization, so if we want to maximize a function, then we simply multiply it by -1 or invert it. This function thus represents a measure of "badness" that we want to minimize. Examples include weight in structural design and direct operating cost in aircraft design. To perform numerical optimization, we require the objective function to be computable for a range of designs. The objective function can be given as an explicit function, or be the result of a complex computational procedure, like the drag coefficient given by computational fluid dynamics.

The choice of objective function is crucial: if the wrong function is used, it does not matter how accurate the computation of the objective is, or how efficient the numerical optimization solution process is. The "optimal" design is likely to be obviously non-optimal from the engineering point of view. A bad choice of objective function is a common mistake in MDO.

Optimization problems are classified with respect to the objective function by determining how the function depends on the design variables (linear, quadratic, or generally nonlinear). In this course, we will concentrate on the general nonlinear case, although as we will see, there is merit in analyzing the quadratic case.

One common misconception is that in engineering design we often want to optimize multiple objectives. As we will see later, it is possible to consider problems with multiple objectives, but this results in a family of optimum designs with different emphasis on the various objectives. In most cases, it makes much more sense to convert these "objectives" into constraints. In the end, we can only make one thing best at a time.

1.2.2 Design Variables

The design variables, x, are the parameters that the optimization algorithm is free to choose in order to minimize the objective function. The optimization problem formulation allows for upper and lower *bounds* for each design variable (also known as *side constraints*). We distinguish these bounds from constraints, since as we will see later, they require a different numerical treatment when solving an optimization problem.

Design variables must be truly independent variables, i.e., in a given analysis of the design, they must be input parameters that remain fixed in the analysis cycle. They must not depend on each other or any other parameter.

Design variables can be broadly classified as quantitative and qualitative. Quantitative variables can be expressed by a number, which could be continuous or discrete. A real variable will result in the continuous case if allowed to vary at least within a certain range. If only discrete values are allowed (e.g., only plates of a certain thickness in a structure are available), then this results in the discrete case. Integer design variables will invariably result in the discrete case. Qualitative variables do not represent quantities, but discrete choices (e.g., the choice of aircraft configuration: monoplane vs. biplane, etc.). In this course, we will focus primarily on continuous design variables.

1.2.3 Constraints

The vast majority of practical design optimization problems have constraints. These are functions of the design variables that we want to restrict in some way. When we restrict a function to being equal to a fixed quantity, we call this an *equality constraint*. When the function is required to be greater than or equal to a certain quantity, we have an *inequality constraint*. The convention we will use for inequality constraints is greater or equal, so we will have to make sure that less or equal constraints are multiplied by -1 before solving the problem. Be aware that the opposite convention is used in some texts or software packages (like Matlab).

As in the case of the objective function, the constraint functions can be linear, quadratic, or nonlinear. We will focus on the nonlinear case, but the linear case will provide the basis for some of the optimization algorithms.

Inequality constraints can be *active* or *inactive* at the optimum point. If inactive, then the corresponding constraint could have been left out of the problem with no change in its solution. In the general case, however, it is difficult to know in advance which constraints are active or not. Constrained optimization is the subject of Chapter 5.

1.2.4 Optimization Problem Statement

Now that we have defined the objective function, design variables, and constraints, we can formalize the optimization problem statement as shown below.

$$\begin{aligned} \text{minimize} \quad & f(x) \\ \text{by varying} \quad & x \in \mathbb{R}^n \\ \text{subject to} \quad & \hat{c}_j(x) = 0, \quad j = 1, 2, \ldots, \hat{m} \\ & c_k(x) \geq 0, \quad k = 1, 2, \ldots, m \end{aligned} \tag{1.1}$$

f : objective function, output (e.g., structural weight).

x : vector of design variables, inputs (e.g., aerodynamic shape); bounds can be set on these variables.

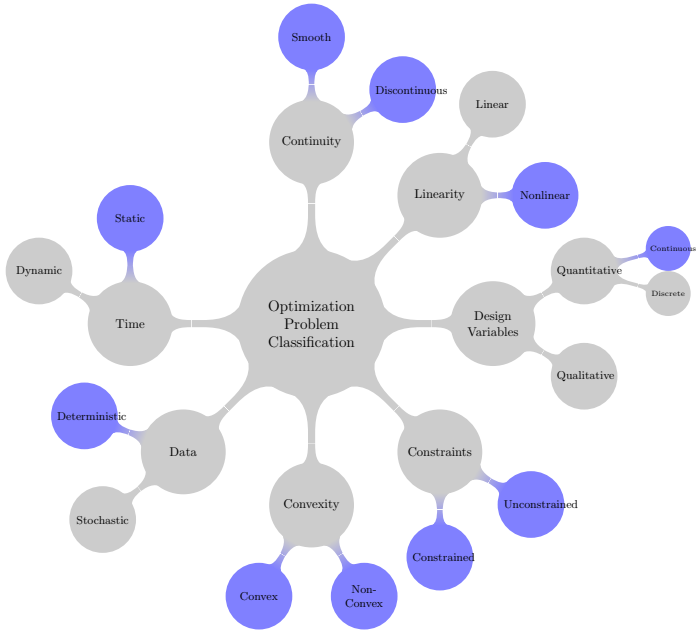


Figure 1.3: Classification of optimization problems; the course focuses on the types highlighted in blue.

$\hat{c}$: vector of equality constraints (e.g., lift); in general these are nonlinear functions of the design variables.

c : vector of inequality constraints (e.g., structural stresses), may also be nonlinear and implicit.

where f is the objective function, x is the vector of n design variables, $\hat{c}$ is the vector of $\hat{m}$ equality constraints and c is the vector of m inequality constraints.

1.2.5 Classification of Optimization Problems

Optimization problems are classified based on the various characteristics of the objective function, constraint functions, and design variables. Fig. 1.3 shows some of the possibilities.

In this course, we will focus on general nonlinear, non-convex, constrained optimization problems with continuous design variables. However, you should be aware that restricting the optimization to certain types and using specialized optimization algorithms can result in a dramatic improvement in the capacity to solve a problem.

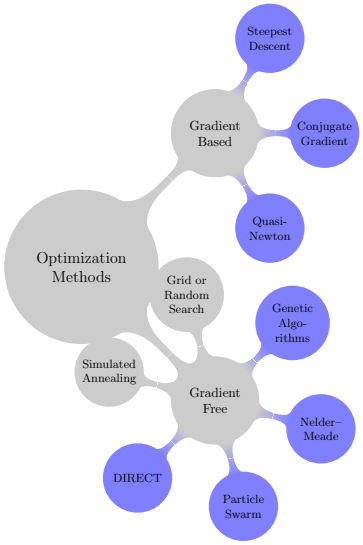


Figure 1.4: Optimization methods for nonlinear problems

1.3 Notation

Matrices and vectors are used heavily throughout the text, and as a consequence we do not use special symbols like arrows or a bold font. We generally adopt the following convention: greek symbols for scalars, lowercase letters for vectors, and uppercase letters for matrices. Some exceptions on scalars versus vectors are contained in Chapter 2 when the equations are one-dimensional.

Other important notations:

- Vectors are column vectors and thus $x^T y$ represents the dot product between vector x and vector y.
- x_i generally refers to the ith element of vector x, whereas x_k represents the entire vector x at iteration k.
- Objective functions are denoted by f, gradients by g, and Hessians by H.
- A star denotes an optimal value, and thus f^* is the optimal objective value, and x^* is the set of design varaible at the optimum.
- f_k is shorthand for $f(x_k)$, or the value of a function at x_k. Similarly, for g_k and H_k.

1.4 Timeline of Historical Developments in Optimization

300 bc: Euclid considers the minimal distance between a point a line, and proves that a square has the greatest area among the rectangles with given total length of edges.

200 bc: Zenodorus works on "Dido's Problem", which involved finding the figure bounded by a line that has the maximum area for a given perimeter.

100 bc: Heron proves that light travels between two points through the path with shortest length when reflecting from a mirror, resulting in an angle of reflection equal to the angle of incidence.

1615: Johannes Kepler finds the optimal dimensions of wine barrel. He also formulated an early version of the "marriage problem" (a classical application of dynamic programming also known as the "secretary problem") when he started to look for his second wife. The problem involved maximizing a utility function based on the balance of virtues and drawbacks of 11 candidates.

1621 W. van Royen Snell discovers the law of refraction. This law follows the more general *principle of least time* (or Fermat's principle), which states that a ray of light going from one point to another will follow the path that takes the least time.

1646: P. de Fermat shows that the gradient of a function is zero at an extreme point.

1695: Isaac Newton solves for the shape of a symmetrical body of revolution that minimizes fluid drag using calculus of variations.

1696: Johann Bernoulli challenges all the mathematicians in the world to find the path of a body subject to gravity that minimizes the travel time between two points of different heights — the *brachistochrone problem*. Bernoulli already had a solution that he kept secret. Five mathematicians respond with solutions: Isaac Newton, Jakob Bernoulli (Johann's brother), Gottfried Leibniz, Ehrenfried Walther von Tschirnhaus, and Guillaume de l'Hôpital. Newton reportedly started solving the problem as soon as he received it, did not sleep that night and took almost 12 hours to solve it, sending back the solution that same day.

1740: L. Euler's publication begins the research on general theory of calculus of variations.

1746: P. L. Maupertuis proposed the *principle of least action*, which unifies various laws of physical motion. This is the precursor of the variation principle of stationary action, which uses calculus of variations and plays a central role in Lagrangian and Hamiltonian classical mechanics.

1784: G. Monge investigates a combinatorial optimization problem known as the *transportation problem*.

1805: Adrien Legendre describes the *method of least squares*, which was used in the prediction of asteroid orbits and curve fitting. Frederich Gauss publishes a rigorous mathematical foundation for the method of least squares and claims he used it to predict the orbit of the asteroid Ceres in 1801. Legendre and Gauss engage in a bitter dispute on who first developed the method.

1815: D. Ricardo publishes the *law of diminishing returns* for land cultivation.

1847: A. L. Cauchy presents the steepest descent methods, the first gradient-based method.

1857: J. W. Gibbs shows that chemical equilibrium is attained when the energy is a minimum.

1902: Gyula Farkas presents an important lemma that is later used in the proof of the Karush–Kuhn–Tucker theorem.

1917: H. Hancock publishes the first text book on optimization.

1932: K. Menger presents a general formulation of the *traveling salesman problem*, one of the most intensively studied problems in optimization.

1939: William Karush derives the necessary conditions for the inequality constrained problem in his Masters thesis. Harold Kuhn and Albert Tucker rediscover these conditions and publish their seminal paper in 1951. These became known as the Karush–Kuhn–Tucker (KKT) conditions.

1939 Leonid Kantorovich develops a technique to solve linear optimization problems after having been given the task of optimizing production in the Soviet government plywood industry.

1947: George Dantzig publishes the simplex algorithm. Dantzig, who worked for the US Air Force, reinvented and developed linear programming further to plan expenditures and returns in order to reduce costs to the army and increase losses to the enemy in World War II. The algorithm was kept secret until its publication.

1947: John von Neumann develops the theory of duality for linear problems.

1949: The first international conference on optimization, the International Symposium on Mathematical Programming, is held in Chicago.

1951: H. Markowitz presents his portfolio theory that is based on quadratic optimization. He receives the Nobel memorial prize in economics in 1990.

1954: L. R. Ford and D. R. Fulkerson research network problems, founding the field of combinatorial optimization.

1957: R. Bellman presents the necessary optimality conditions for dynamic programming problems. The Bellman equation was first applied to engineering control theory, and subsequently became an important principle in the development of economic theory.

1959: Davidon develops the first quasi-Newton method for solving nonlinear optimization problems. Fletcher and Powell publish further developments in 1963.

1960: Zoutendijk presents the methods of feasible directions to generalize the Simplex method for nonlinear programs. Rosen, Wolfe, and Powell develop similar ideas.

1963: Wilson invents the *sequential quadratic programming method* for the first time. Han reinvents it in 1975 and Powell does the same in 1977.

1975: Pironneau publishes a seminal paper on aerodynamic shape optimization, which first proposes the use of adjoint methods for sensitivity analysis [11].

1975: John Holland proposed the first genetic algorithm.

1977: Raphael Haftka publishes one of the first multidisciplinary design optimization (MDO) applications, in a paper entitled "Optimization of flexible wing structures subject to strength and induced drag constraints" [2].

1979: Kachiyan proposes the first polynomial time algorithm for linear problems. The New York Times publishes the front headline "A Soviet Discovery Rocks World of Mathematics", saying, "A surprise discovery by an obscure Soviet mathematician has rocked the world of mathematics and computer analysis ... Apart from its profound theoretical interest, the new discovery may be applicable in weather prediction, complicated industrial processes, petroleum refining, the scheduling of workers at large factories ... the theory of secret codes could eventually be affected by the Russian discovery, and this fact has obvious importance to intelligence

agencies everywhere." In 1975, Kantorovich and T.C. Koopmans receive the Nobel memorial prize of economics for their contributions on linear programming.

1984: Narendra Karmarkar starts the age of interior point methods by proposing a more efficient algorithm for solving linear problems. In a particular application in communications network optimization, the solution time was reduced from weeks to days, enabling faster business and policy decisions. Karmarkar's algorithm stimulated the development of several other interior point methods, some of which are used in current codes for solving linear programs.

1985: The first conference in MDO, the Multidisciplinary Analysis and Optimization (MA&O) conference, takes place.

1988: Jameson develops adjoint-based aerodynamic shape optimization for computational fluid dynamics (CFD).

1995: Kennedy and Eberhart propose the particle swarm optimization algorithm.

1.5 Practical Applications

In this section a number of application problems are highlighted. These examples are selected to highlight a range of optimization strategies and features. Only basic details are discussed in this text, but references to full details are provided for the interested reader.

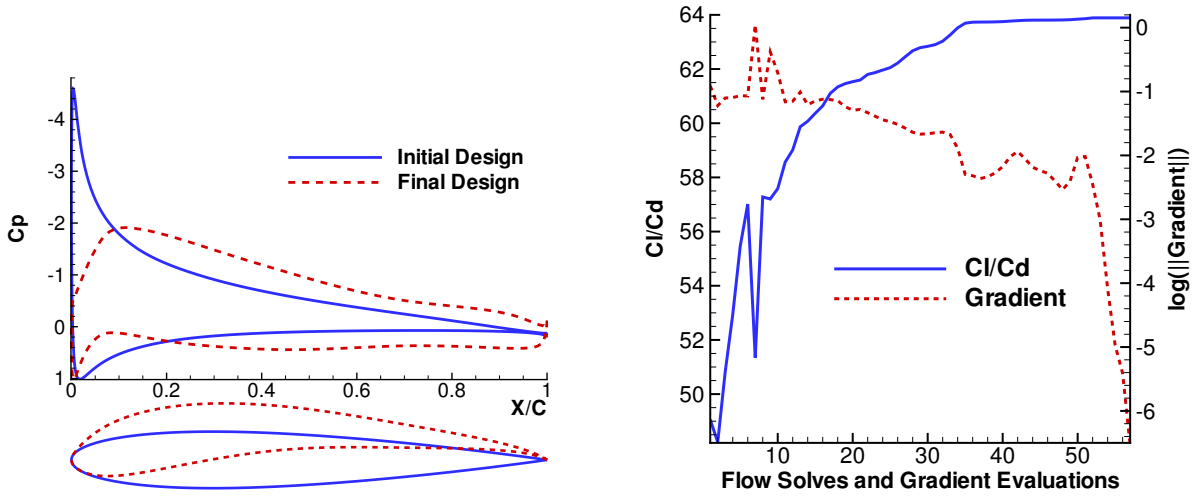
1.5.1 Airfoil Design

This airfoil design problem demonstrates the use of multipoint optimization. Aerodynamic shape optimization emerged after computational fluid dynamics (CFD) started to be used more routinely. Once aerodynamicists were able to analyze arbitrary shapes, the natural next step was to devise algorithms that improved the shape automatically [4, 5]. Initially, 2D problems were tackled, resulting in airfoil shape optimization. Today, full configurations can be optimized not only considering aerodynamics [12, 13], but also aerodynamics and structures [9].

The airfoil design optimization example we show here is due to Nemec et al. [10]. The solution shown in Fig. 1.5 corresponds to the maximization of the lift-to-drag ratio of the airfoil with respect to 9 shape variables and angle of attack, subject to airfoil thickness constraints. Figure 1.5b shows the convergence history of the objective on a linear scale and the norm of the gradient on a log scale. The figure highlights the importance of using relevant metrics for convergence criteria plotted on a log scale (discussed further in Chapter 2). Although the objective makes only small changes after about 35 iterations, the corresponding changes in the design variables are still significant and good convergence is not achieved until around 54 iterations.

Instead of maximizing the lift-to-drag ratio, another possible objective function is the drag coefficient. In this case, we must impose a lift constraint otherwise the optimal solution would be zero drag and zero lift. Fig. 1.6 shows the results for a drag minimization problem. We can see that the optimized result minimizes the drag at the specified condition to a fault and exhibits particularly bad off-design performance. This is a typical example of the optimizer exploiting a weakness in the problem formulation.

To improve off-design performance, we can include the performance of multiple flight conditions in the objective function. This is the case for the problem shown in Fig. 1.7, in which four flight conditions where considered.



(a) Airfoil shape and pressure coefficient distributions

(b) Convergence history

Figure 1.5: Airfoil shapes and corresponding pressure coefficient distributions for lift-to-drag maximization case.

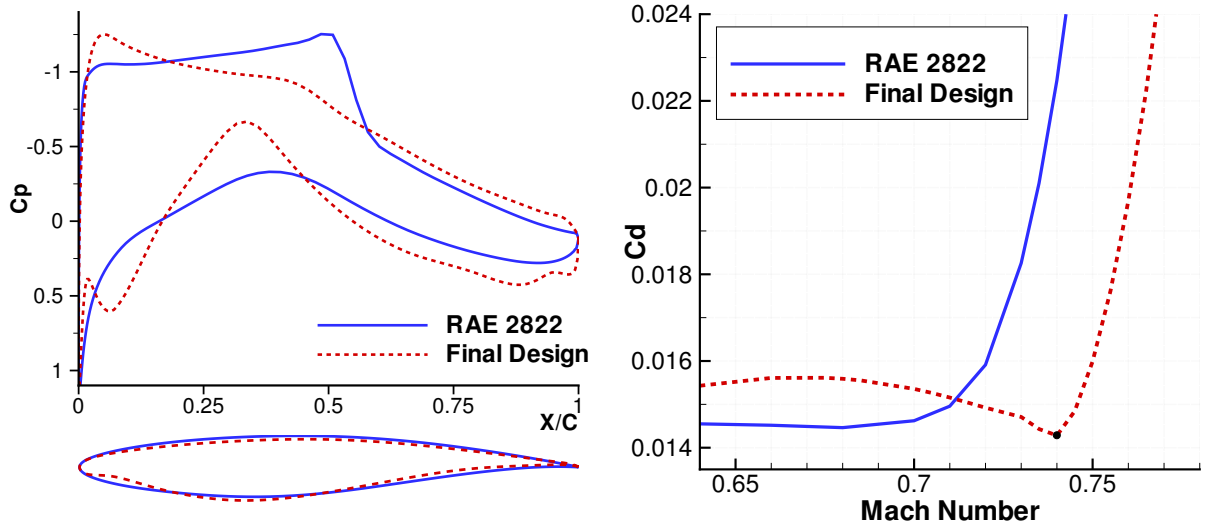


Figure 1.6: Airfoil shapes and corresponding pressure coefficient distributions for single point drag minimization.

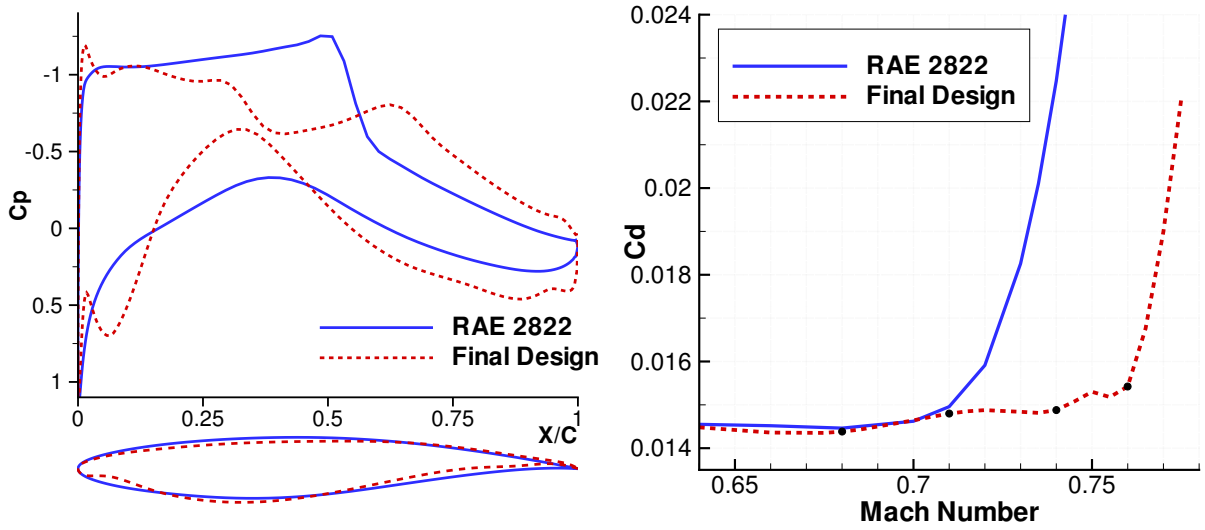
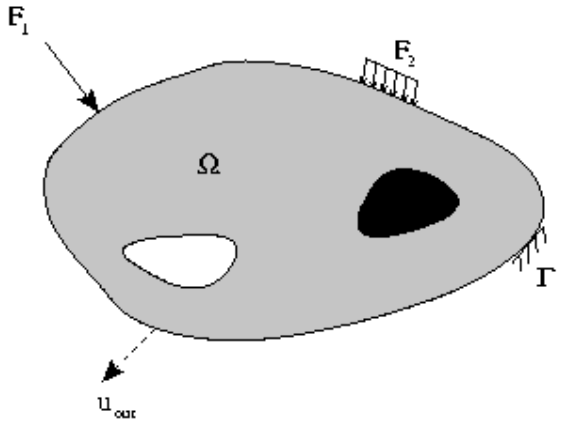


Figure 1.7: Airfoil shapes and corresponding pressure coefficient distributions for multipoint drag minimization.

1.5.2 Structural Topology Optimization

This example is from recent work by James and Martins [3]. The objective of structural topology optimization is to find the shape and *topology* of a structure that has the minimum compliance (maximum stiffness) for a given loading condition.



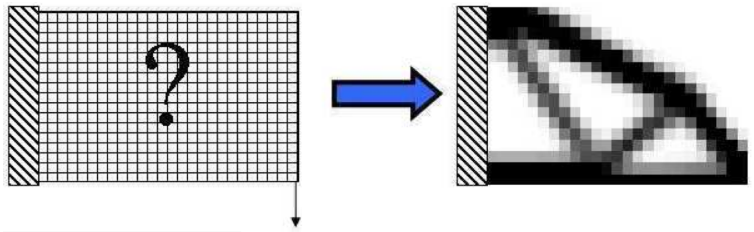


Figure 1.8: Structural topology optimization process

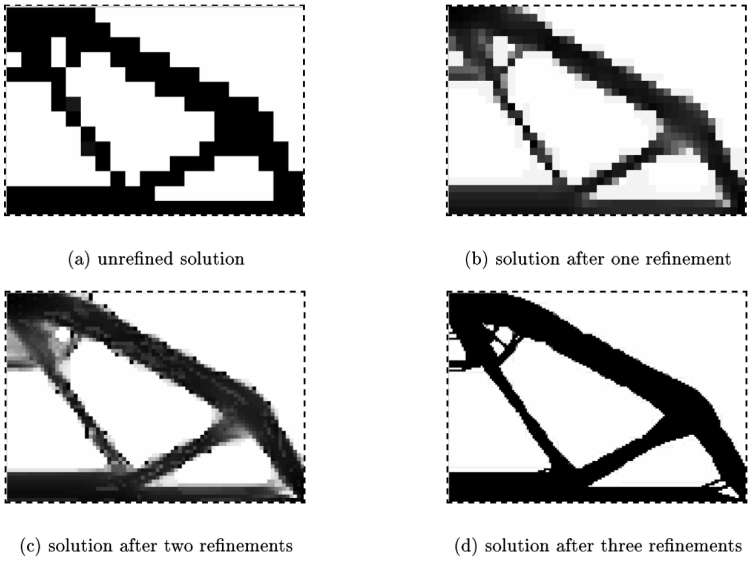


Figure 1.9: Optimized topology showing four different levels of refinement

1.5.3 Aircraft Design with Minimum Environmental Impact

The environmental impact of aircraft has been a popular topic in the last few years. In this example, Antoine and Kroo [1] show the trade-offs between cost, noise, and greenhouse gas emissions, by solving a number of multiobjective problems. Figure 1.10 highlights the multidisciplinary nature of this problem. Figure 1.11 contains a *Pareto front*, which is a curve (or surface) containing an infinite number of "optimal" designs. These curves are meant to show trade-offs in cost versus fuel, cost versus NO_x emissions, and cost versus noise. If we wanted to do design then the "objectives" for NO_x emissions and noise would be more appropriate formulated as constraints. Additionally, fuel burn is already accounted for in costs and need not be a separate "objective". However, Pareto fronts and Pareto surfaces can be useful in visualizing sensitivities and comparing a family of designs. For example, we can quantify the increase in flying costs if we choose to enforce more stringent noise standards, or realize that a small increase in cost (by slowing down and unsweeping the wing) could significantly reduce NO_x emissions.

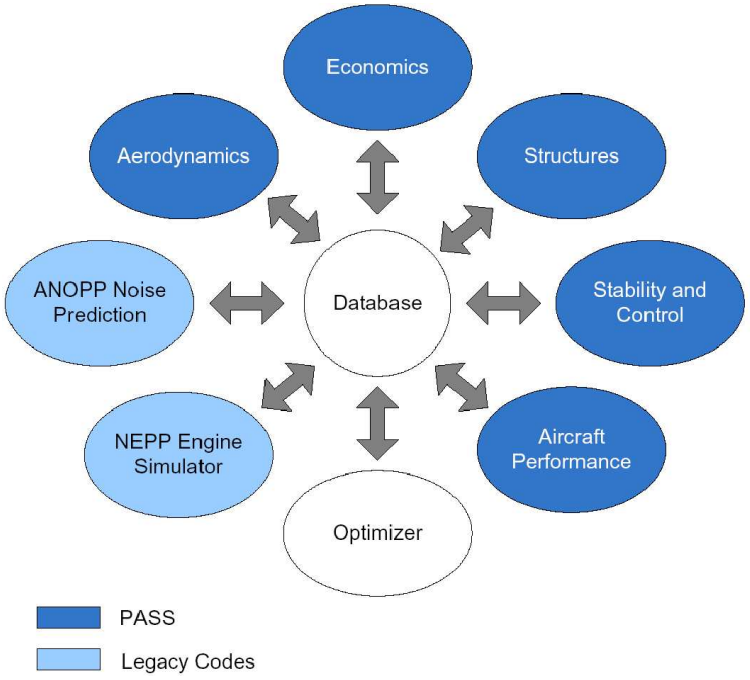


Figure 1.10: Analysis and optimization framework. PASS stands for Program for Aircraft Synthesis Studies and is an in-house aircraft design code.

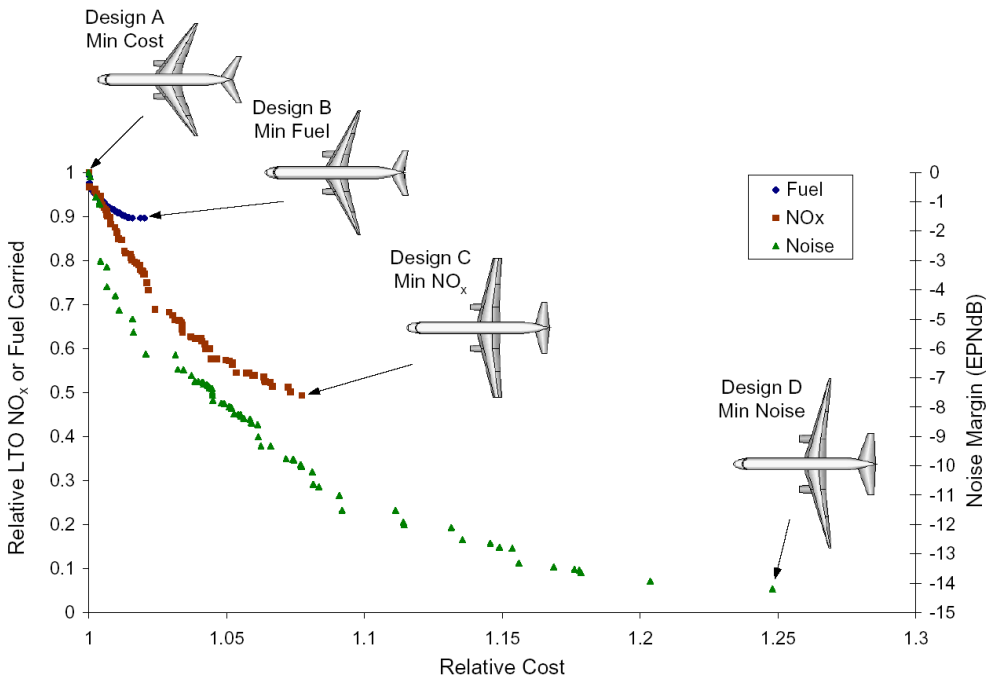


Figure 1.11: Pareto fronts of fuel carried, emissions and noise vs. operating cost

1.5.4 Aerodynamic Design of a Natural Laminar Flow Supersonic Business Jet

The goal of this project was to aid the design of a natural laminar flow wing for a supersonic business jet, a concept that is being developed by Aerion Corporation and Desktop Aeronautics [8].

In this work, a CFD Euler code was combined with a boundary-layer solver to compute the flow on a wing-body. The fuselage spoils the laminar flow that can normally be maintained on a thin, low sweep wing in supersonic flow. The goal is to reshape the fuselage at the wing-body junction to maximize the extent of laminar flow on the wing. Three design variables were used initially, with quadratic response surfaces and a trust region update algorithm.

The baseline design, whose solution is shown in Fig. 1.14, is a Sears–Haack body with wing. This results in early transition (the white areas in the boundary-layer solution). N^* is the measure of laminar instability, with 1.0 (white) being the prediction of transition. The flow is then turbulent from the first occurrence of $N^* = 1$ to the trailing edge irrespective of further values of N^*.

With only three design variables (the crosses on the fuselage outline that sit on the wing) and two iterations (not even near a converged optimization) the improvement is dramatic. With five design variables, and a few more trust-region update cycles, a better solution is found.

The boundary layer is much farther from transition to turbulent flow as can be seen by comparing the green and yellow colors on this wing with the red and violet colors in the three variables case. Also notice how subtle the reshaping of the fuselage is. This is typical of aerodynamic shape optimization: small changes in shape have a big impact in the design.

Figure 1.12: Supersonic business jet configuration

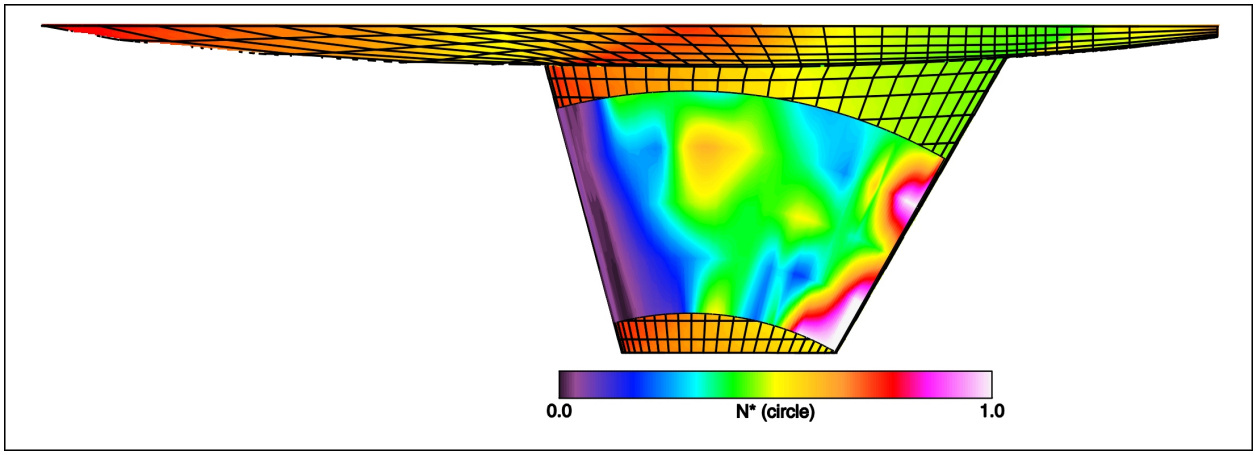


Figure 1.13: Aerodynamic analysis showing the boundary-layer solution superimposed on the Euler pressures

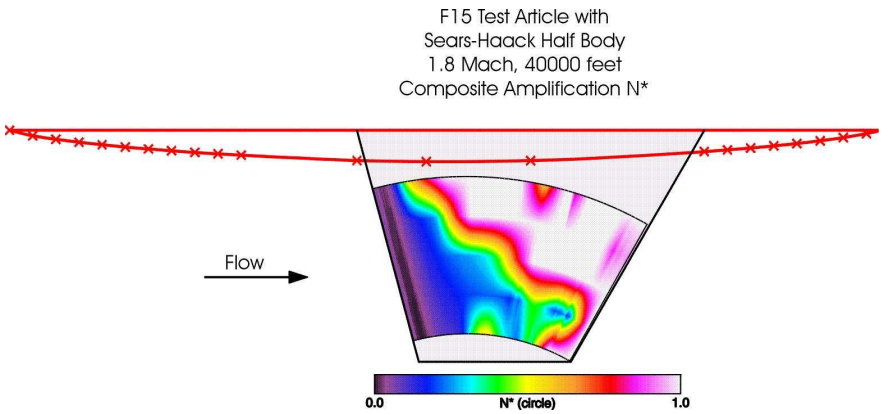


Figure 1.14: Baseline design. The colors on the wing show a measure of laminar instability: White areas represent transition to turbulent boundary layer.

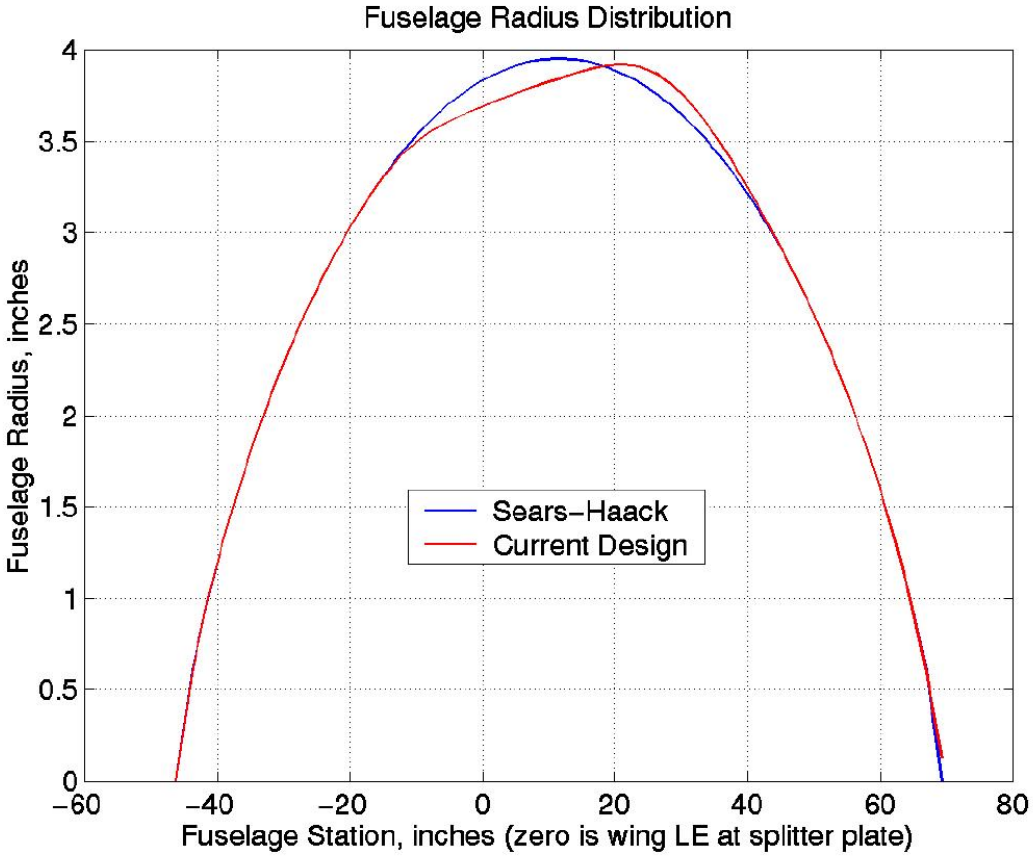


Figure 1.15: Fuselage shape optimization with three shape variables. From the nose at left, to the tail at right, this is the radius of the original (blue) and re-designed (red) fuselage after two iterations.

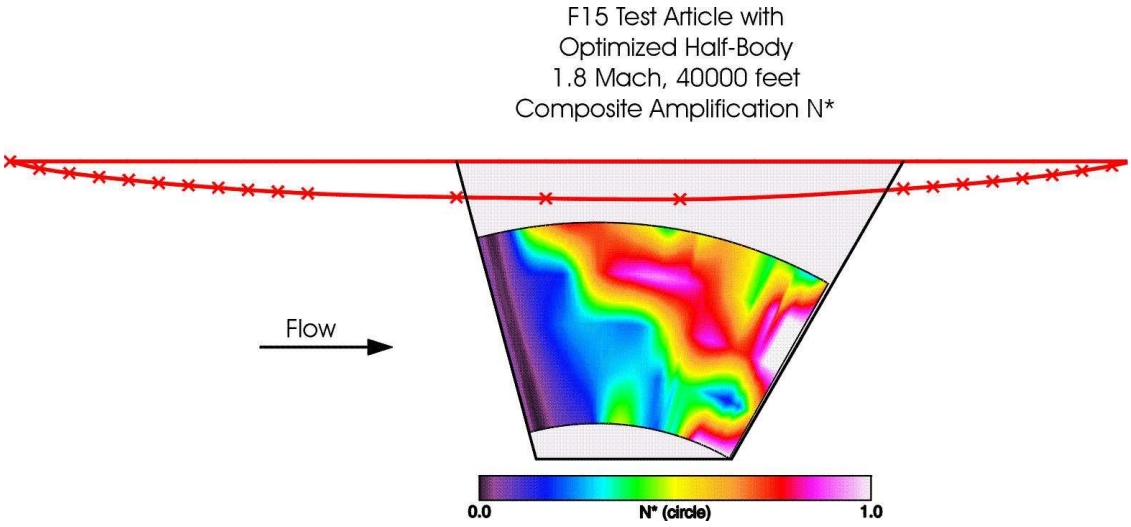


Figure 1.16: Laminar instability for fuselage optimized with three design variables

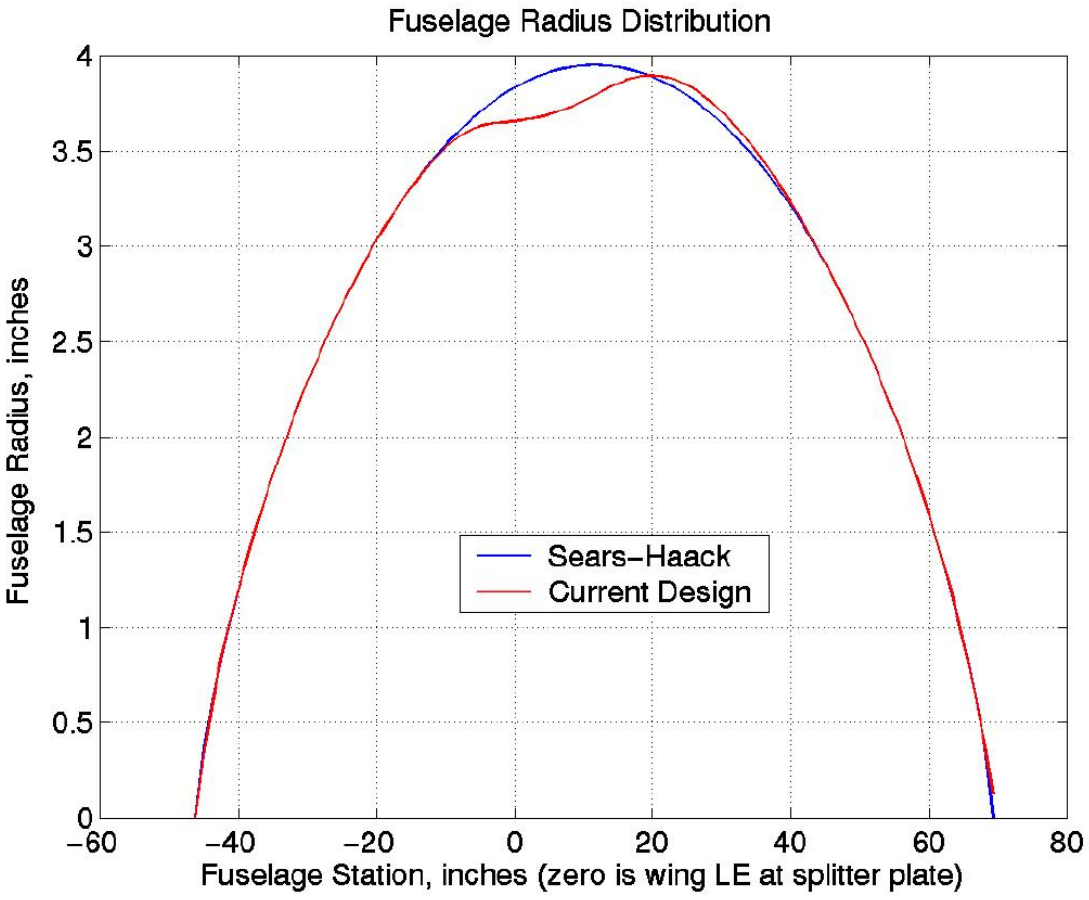


Figure 1.17: Fuselage shape optimization with five shape variables.

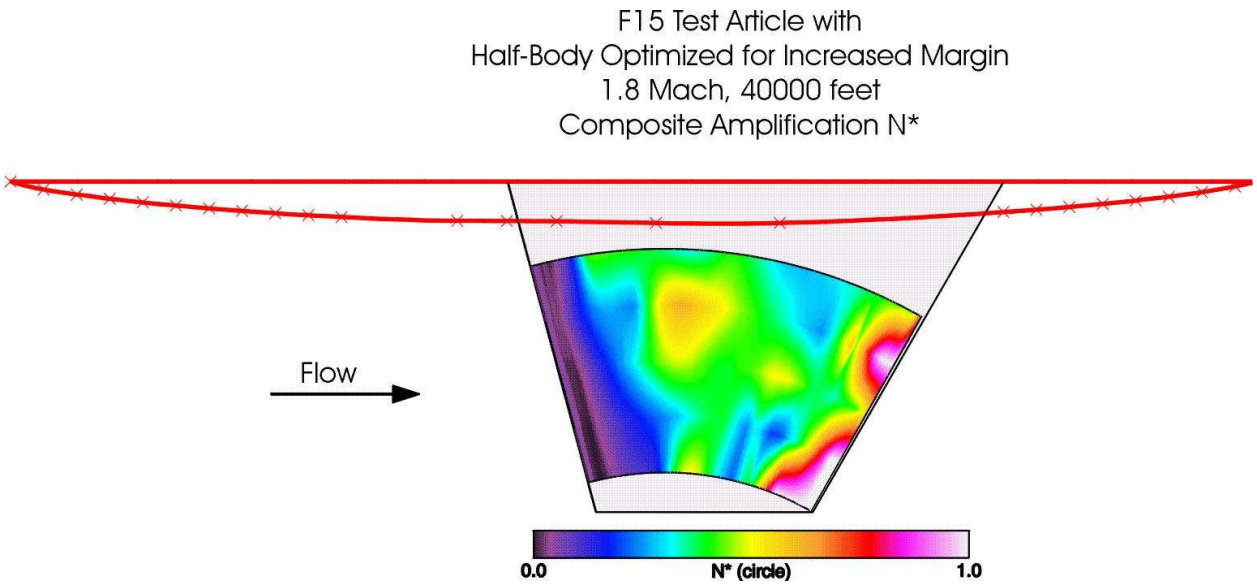
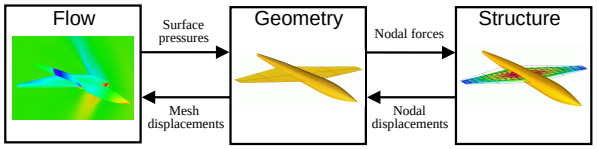
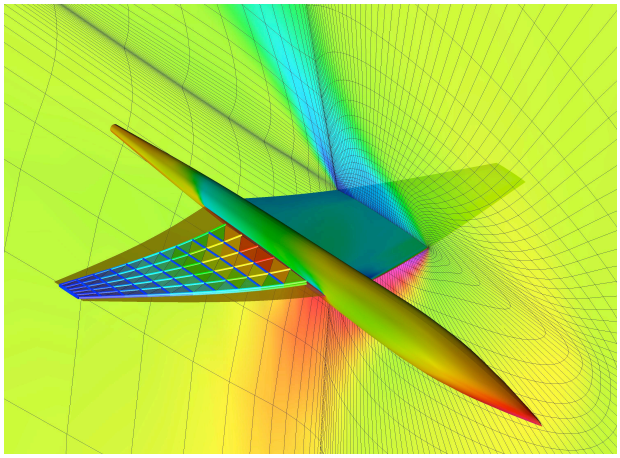


Figure 1.18: Laminar instability for fuselage optimized with 5 design variables

1.5.5 Aerostructural Design of a Supersonic Business Jet

See Martins et al. [9] for details.





- Aerodynamics: a parallel, multiblock Euler flow solver.
- Structures: detailed finite element model with plates and trusses.
- Coupling: high-fidelity, consistent and conservative.
- Geometry: centralized database for exchanges (jig shape, pressure distributions, displacements)
- Coupled-adjoint sensitivity analysis: aerodynamic and structural design variables.

Supersonic business jet specification:

- Natural laminar flow
- Cruise at Mach = 1.5
- Range = 5,300nm

- 1 count of drag = 310 lbs of weight

Objective function to be minimized:

$$I = \alpha C_D + \beta W \tag{1.2}$$

where C_D is that of the cruise condition. The structural constraint is an aggregation of the stress constraints at a maneuver condition, i.e.,

$$\text{KS}(\sigma_m) \geq 0 \tag{1.3}$$

The design variables are external shape and internal structural sizes.

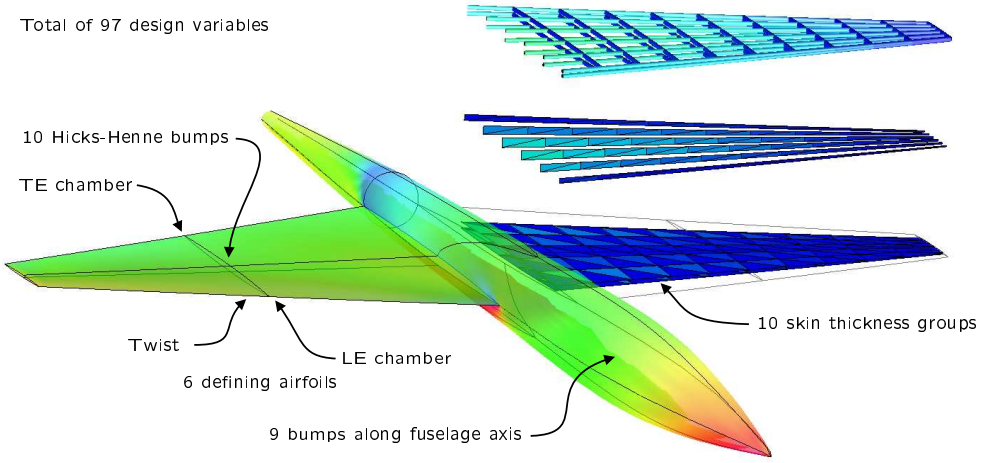


Figure 1.19: Baseline configuration and design variables

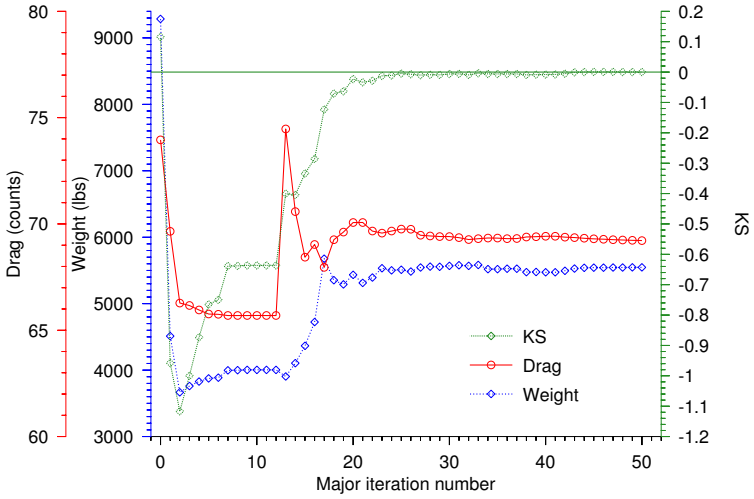


Figure 1.20: Optimization history

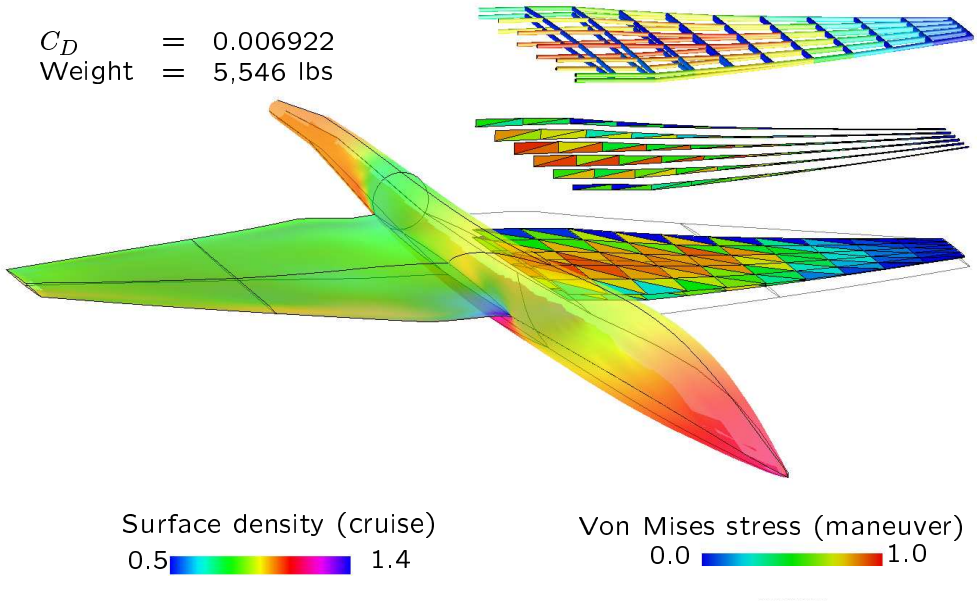


Figure 1.21: Optimized design

1.5.6 Aerostructural Shape Optimization of Wind Turbine Blades Considering Site-Specific Winds

See Kenway and Martins [7] for details.

- What effect does a location's particular wind distribution have on optimum design?
- Two sites are chosen to represent the two differing environments:
- Direct simulation of the total mechanical output

Objective: Minimize the cost of energy, COE = $\frac{C}{\text{AEP}}$
Design Variables:

Design Variable	Count	Lower Limit	Upper Limit
Chord	4	.05 m	.40 m
Twists	4	-75 deg	75 deg
W_{spar}	4	4%	30%
t_{spar}	4	0.3 mm	10mm
t_{foil}	3	6%	20%
Ω	varies (12)	7.5 rad/s	14.7 rad/s

Constraints:

Constraint	Minimum	Maximum
Stress	-	40MPa
Spar Mass	-	3.7kg
Surface Area	-	0.83m^2
Power	-	5000 W
Geometry	0.5mm	-

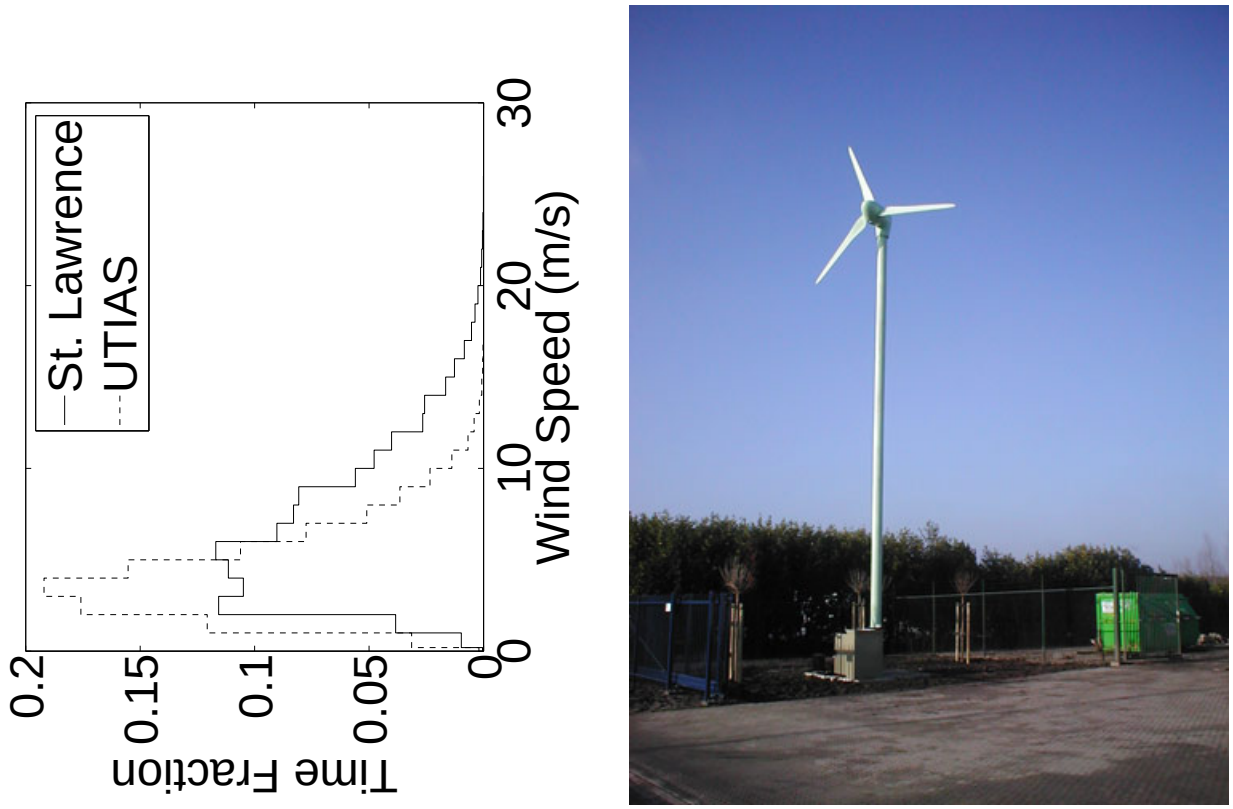
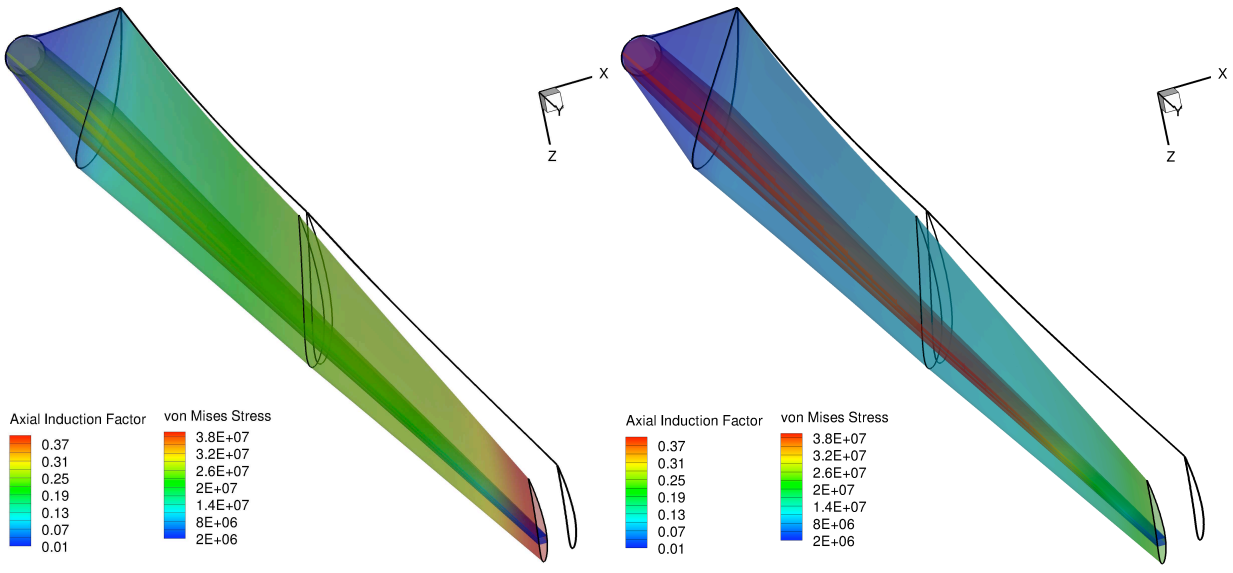
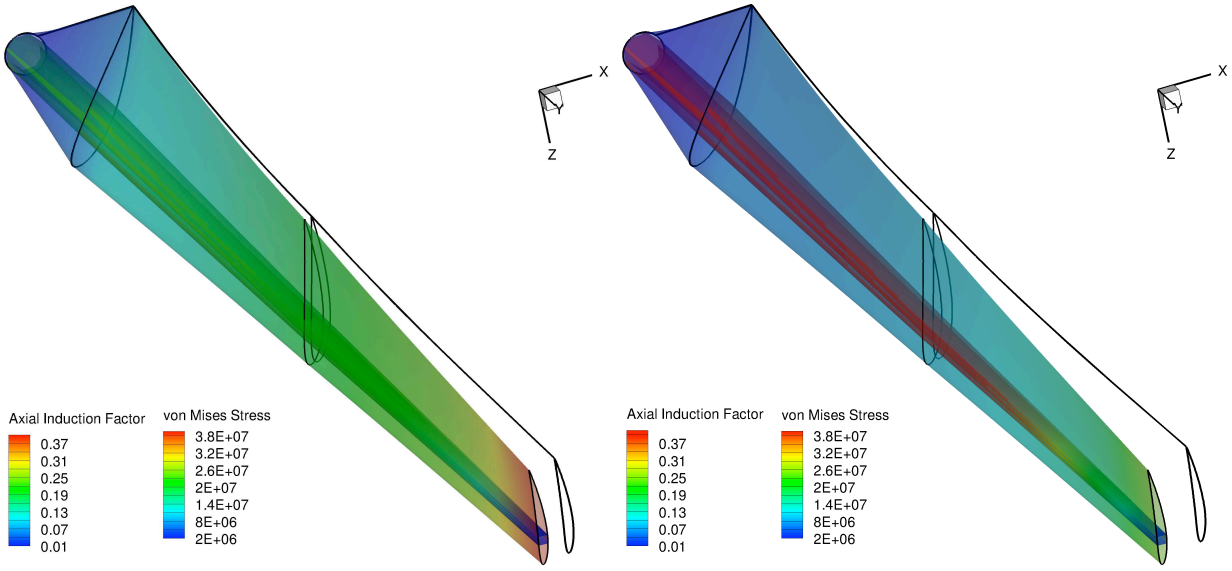


Figure 1.22: Wind speed distributions (left) and the Wes5 Tulipo wind turbine (right).

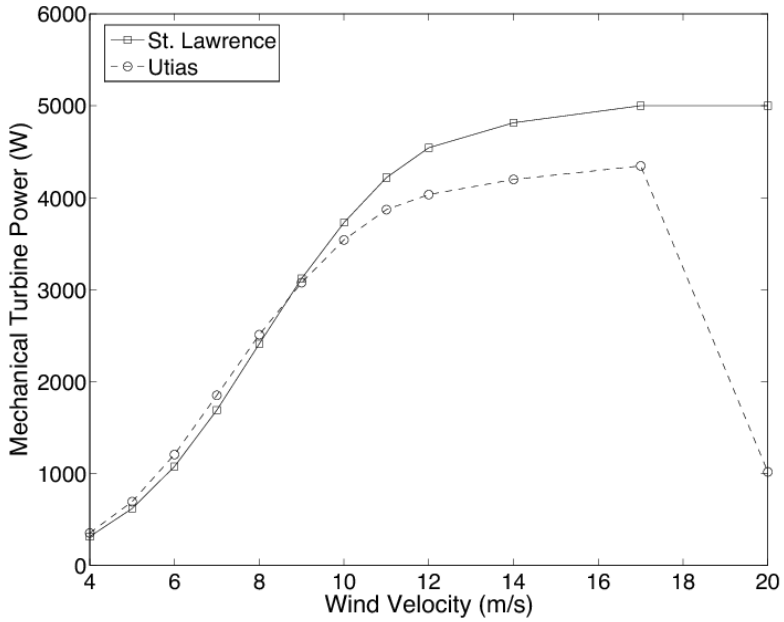


7m/s 12m/s

Location	$\bar{P}_{\text{init}}$ (W)	$\bar{P}_{\text{opt}}$ (W)	$\bar{P}_{\text{other-opt}}$ (W)	Site-specific increase
St. Lawrence	1566.1	1984.5	1905.1	4.17%
UTIAS	660.2	853.3	826.0	3.31%



7 m/s 12 m/s



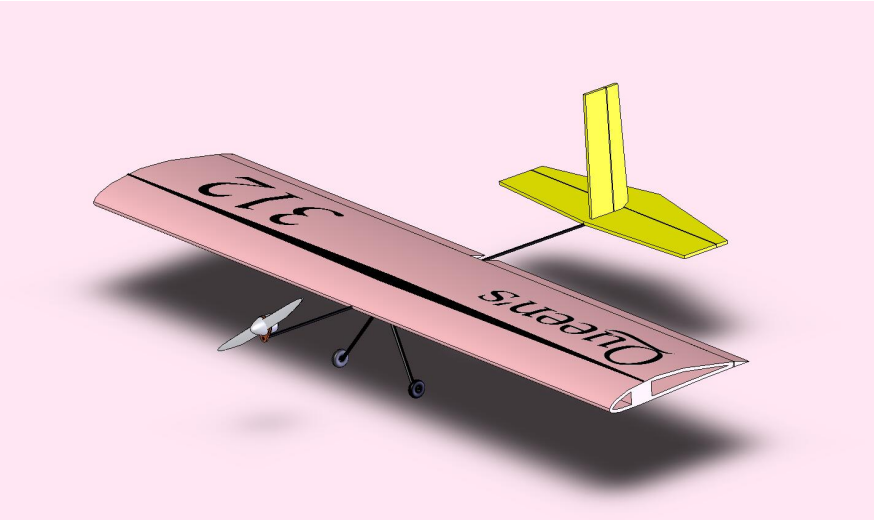
1.5.7 MDO of an Airplane for the SAE Micro-Class Aero Design Competition

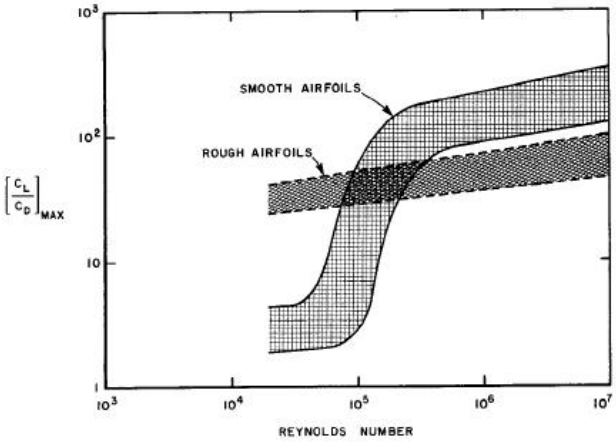
See Kenway [6] for details.

Maximize: Score $= 13 \times \text{PF} \times (10 - \text{EW})$

Where: PF = $\frac{\text{Payload}}{\text{EW+Payload}}$

EW = Empty Weight in lbs



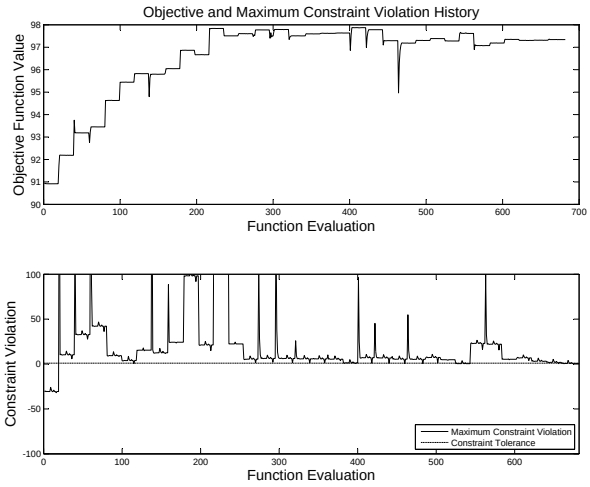


L/D as a function of Reynolds Number

- XFOIL: 2D panel method
- AVL: 3D panel method
- Matlab: FE method for structure and mass predication
- MotoCalc for propulsion

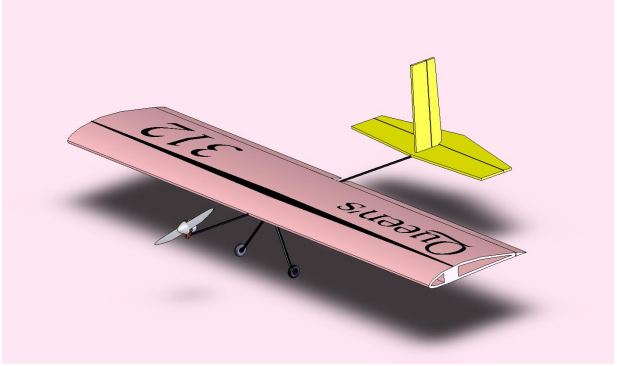
SubGroup	Variable	Start Point	Lower Bound	Upper Bound	Units
Wing	Span	36	24	45	inches
	Aspect Ratio	3.6	3	8	-
	Taper Ratio	1	0.5	1	-
	Root Airfoil Camber	4	2	5	%
	Root Airfoil Thickness	14	8	15	%
	Root Carbon	0.75	0.1	3	inches
	Tip Airfoil Camber	4	2	5	%
	Tip Airfoil Thickness	14	8	15	%
	Tip Carbon	.25	0.1	3	inches
Tail Feathers	Moment Arm	18	12	45	inches
	H-Span	15	8	20	inches
	H-Aspect Ratio	3.35	2	4	-
	H-Taper Ratio	0.7	0.5	1	-
	V-Span	7	7	20	inches
	V-Aspect Ratio	1.31	1.5	3	-
	V-Taper Ratio	1.31	0.5	1	-
Fuselage	Motor Moment Arm	13.75	4	20	inches
	Fuselage Spline Diameter	0.25	0.05	0.5	inches

Constraint	Value
Balance	$23\% < X_{cg} < 27\%$
Static Margin	$> 15\%$
Wing Deflection	< 1”
Tail Deflection	$< 3/8$”
Elevator C_L	> -0.5
Elevator Deflection	$< 20°$



- Optimization increased score over “good” design by 7%
- The ultimate experimental validation!

Objective and constraint history

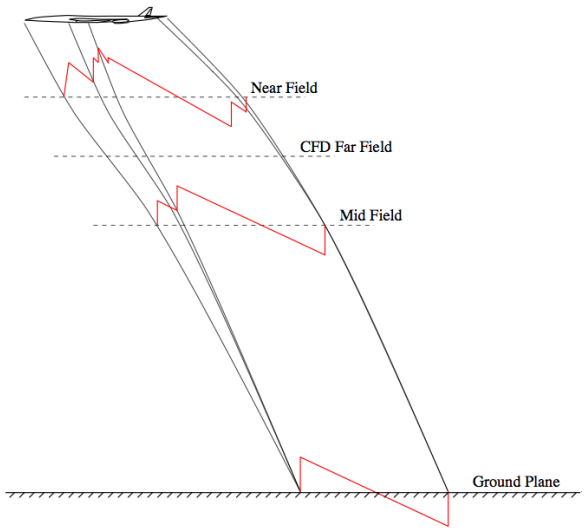


- Official Aero Design West Score: 96.3
- MDO Best Score Result: 97.3
- MDO works!
- First place in Micro class and lowest empty weight

1.5.8 MDO of Supersonic Low-boom Designs

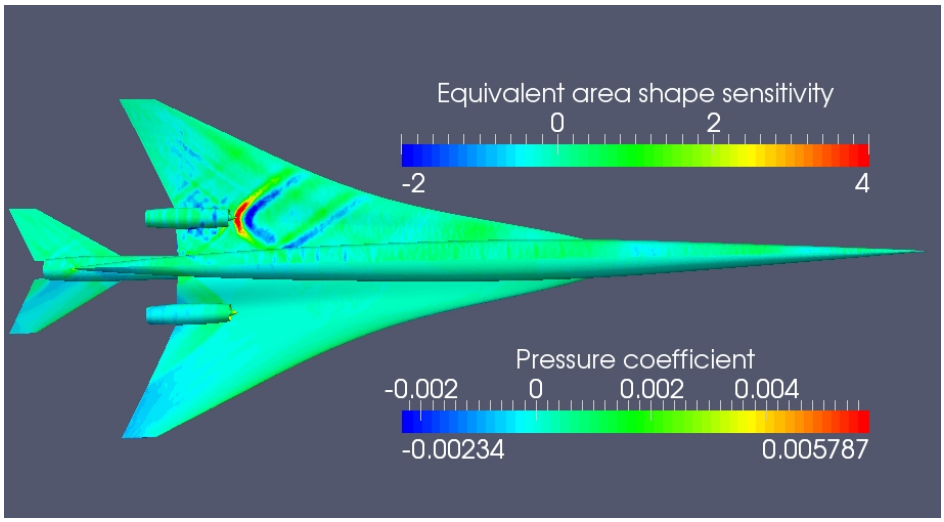
The goal in low-boom design is to reduce the strength of the sonic boom sufficiently to permit supersonic flight overland. The idea is to make subtle changes to the shape of the aircraft that lead to changes in the sonic boom at ground level. This is a very challenging MDO problem. For example, the shape has a direct impact on both the aerodynamic performance and the sonic-boom. In addition, the strength of the sonic boom is related to the weight of the aircraft and length of the aircraft.

One idea for low-boom design is to find (somehow) a target equivalent-area distribution that produces an acceptable ground level boom. Subsequently, we can use CFD-based inverse design to shape an aircraft to match the desired target equivalent area. This is the approach taken in Palacios [**?**].



Propagation of the sonic boom from the aircraft near field to the ground.

Early geometry for the Nasa/Lockheed N+2 configuration.



Pressure field (lower half) and equivalent-area shape sensitivity fields.

Bibliography

[1] Nicolas E. Antoine and Ilan M. Kroo. Aircraft optimization for minimal environmental impact. *Journal of Aircraft*, 41(4):790–797, 2004.

[2] Raphael T. Haftka. Optimization of flexible wing structures subject to strength and induced drag constraints. *AIAA Journal*, 14(8):1106–1977, 1977.

[3] Kai James, Jorn S. Hansen, and Joaquim R. R. A. Martins. Structural topology optimization for multiple load cases using a dynamic aggregation technique. *Engineering Optimization*, 41 (12):1103–1118, December 2009. doi:10.1080/03052150902926827.

[4] A. Jameson. Aerodynamic design via control theory. *Journal of Scientific Computing*, 3(3): 233–260, sep 1988.

[5] Antony Jameson. Computational aerodynamics for aircraft design. *Science*, 245:361–371, 1989.

[6] Gaetan Kenway. Multidisciplinary design optimization of small-scale UAVs. Undergraduate thesis, Queen's University, Kingston, Ontario, Canada, March 2007.

[7] Gaetan Kenway and Joaquim R. R. A. Martins. Aerostructural shape optimization of wind turbine blades considering site-specific winds. In *Proceedings of the 12th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Victoria, BC, September 2008. AIAA 2008-6025.

[8] Ilan M. Kroo, Richard Tracy, James Chase, and Peter Sturdza. Natural laminar flow for quiet and efficient supersonic aircraft. *AIAA Paper* 2002-0146, 2002.

[9] Joaquim R. R. A. Martins, Juan J. Alonso, and James J. Reuther. High-fidelity aerostructural design optimization of a supersonic business jet. *Journal of Aircraft*, 41(3):523–530, 2004. doi:10.2514/1.11478.

[10] Marian Nemec, David W. Zingg, and Thomas H. Pulliam. Multi-point and multi-objective aerodynamic shape optimization. *AIAA Paper* 2002-5548, 2002.

[11] O. Pironneau. On optimum design in fluid mechanics. *Journal of Fluid Mechanics*, 64:97–110, 1974.

[12] James J. Reuther, Antony Jameson, Juan J. Alonso, , Mark J. Rimlinger, and David Saunders. Constrained multipoint aerodynamic shape optimization using an adjoint formulation and parallel computers, part 1. *Journal of Aircraft*, 36(1):51–60, 1999.

[13] James J. Reuther, Antony Jameson, Juan J. Alonso, , Mark J. Rimlinger, and David Saunders. Constrained multipoint aerodynamic shape optimization using an adjoint formulation and parallel computers, part 2. *Journal of Aircraft*, 36(1):61–74, 1999.

Chapter 2

Line Search Techniques

2.1 Introduction

Most practical optimization problems involve many variables, so the study of single variable minimization may seem academic. However, the optimization of multivariable functions can be broken into two parts: 1) finding a suitable search direction and 2) minimizing along that direction. The second part of this strategy, the *line search*, is our motivation for studying single variable minimization.

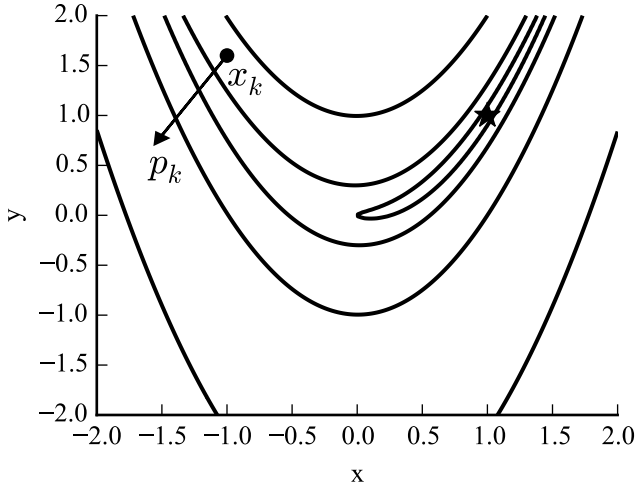


Figure 2.1: Contour plot of the Rosenbrock function with the location of the minimum denoted by a star. A sample starting point x_k with direction vector p_k is overlaid.

Consider the 2D optimization problem shown in Fig. 2.1, where the true minimum is located at the star. Our current evaluation point is x_k and we have a search direction p_k. In Chapter 3 we will discuss how to choose p_k, but for now we assume that it is given. Our task in a line search is to decide how far to move in the p_k direction. Once we move to the new location we repeat the process (choose a new search direction and perform a line search) until we satisfy some convergence criteria and reach the optimum.

A logical choice in selecting how far to move in the p_k direction is to choose the step size that minimizes the function value in that direction. One-dimensional optimization methods can help us do that. However, it is not clear that this is a desirable approach. The minimum along our line will

in general have nothing to do with the minimum of the function we are actually interested in. If we knew the direction that would point us towards the minimum in the n-dimensional space then finding the optimal step on the line would be a great choice. Unfortunately, we do not know the direction that leads to the minimum, and in fact know nothing about the underlying function except in points we have already visited during the optimization. In general, our search directions will not point us to the minimum, and may even be pointing away from our eventually goal. Because it is desirable to take as few function calls as possible, we will instead be satisfied with a step that is "good enough" so that we can move forward and update our search direction. In Fig. 2.1, for example, the p_k is a good direction because it is a descent direction. We will want to make some progress in our descent direction, but it will be wasteful to spend too much time trying to find the exact minimum along that path.

Even though we are not necessarily interested in the optimal value along our line search, 1D optimization methods are still useful to help us find a satisfactory step size quickly. In other words, we will use 1D optimization methods, but rather than running to convergence we will run them until we satisfy a less stringent set of criteria called the Wolfe conditions. The remainder of this chapter introduces one-dimensional optimality conditions, describes various root finding and minimization methods that are useful in line searches, and defines the Wolfe conditions used to terminate a line search.

2.2 Optimality Conditions

We can classify a minimum as a:

1. Strong local minimum
2. Weak local minimum
3. Global minimum

As we will see, Taylor's theorem is useful for identifying local minima. Taylor's theorem states that if $f(x)$ is n times differentiable, then there exists $\theta \in (0, 1)$ such that

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \cdots + \frac{1}{(n-1)!}h^{n-1}f^{n-1}(x) + \underbrace{\frac{1}{n!}h^n f^n(x+\theta h)}_{\mathcal{O}(h^n)} \tag{2.1}$$

Now we assume that f is twice-continuously differentiable and that a minimum of f exists at x^*. Then, using $n = 2$ and $x = x^*$ in Taylor's theorem we obtain,

$$f(x^* + \varepsilon) = f(x^*) + \varepsilon f'(x^*) + \frac{1}{2}\varepsilon^2 f''(x^* + \theta\varepsilon) \tag{2.2}$$

For x^* to be a local minimizer, we require that $f(x^* + \varepsilon) \geq f(x^*)$ for $\varepsilon \in [-\delta, \delta]$, where δ is a positive number. Given this definition, and the Taylor series expansion (2.2), for $f(x^*)$ to be a local minimum, we require

$$\varepsilon f'(x^*) + \frac{1}{2}\varepsilon^2 f''(x^* + \theta\varepsilon) \geq 0 \tag{2.3}$$

For any finite values of $f'(x^*)$ and $f''(x^*)$, we can always chose a ε small enough such that $\varepsilon f'(x^*) \gg \frac{1}{2}\varepsilon^2 f''(x^*)$. Therefore, we must consider the first derivative term to establish under which conditions we can satisfy the inequality (2.3).

For $\varepsilon f'(x^*)$ to be non-negative, then $f'(x^*) = 0$, because the sign of ε is arbitrary. This is the *first-order optimality condition*. A point that satisfies the first-order optimality condition is called a *stationary point*.

Because the first derivative term is zero, we have to consider the second derivative term. This term must be non-negative for a local minimum at x^*. Since ε^2 is always positive, then we require that $f''(x^*) \geq 0$. This is the *second-order optimality condition*. Terms higher than second order can always be made smaller than the second-order term by choosing a small enough ε.

Thus the *necessary conditions* for a local minimum are:

$$f'(x^*) = 0; \qquad f''(x^*) \geq 0 \tag{2.4}$$

If $f(x^* + \varepsilon) > f(x^*)$, and the "greater than or equal to" in the required inequality (2.3) can be shown to be greater than zero (as opposed to greater or equal), then we have a *strong* local minimum. Thus, *sufficient conditions* for a strong local minimum are:

$$f'(x^*) = 0; \qquad f''(x^*) > 0 \tag{2.5}$$

The optimality conditions can be used to:

1. Verify that a point is a minimum (sufficient conditions).
2. Realize that a point is not a minimum (necessary conditions).
3. Define equations that can be solved to find a minimum.

Gradient-based minimization methods find a local minima by finding points that satisfy the optimality conditions.

2.3 Convergence Rate

Optimization algorithms compute a sequence of approximate solutions that we hope converges to the solution. Two questions are important when considering an algorithm:

- Does it converge?
- How fast does it converge?

Suppose you we have a sequence of points x_k $(k = 1, 2, \ldots)$ converging to a solution x^*. For a convergent sequence, we have

$$\lim_{k\to\infty} x_k - x^* = 0 \tag{2.6}$$

The *rate of convergence* is a measure of how fast an iterative method converges to the numerical solution. An iterative method is said to converge with order r when r is the largest positive number such that

$$0 \leq \lim_{k\to\infty} \frac{\|x_{k+1} - x^*\|}{\|x_k - x^*\|^r} < \infty \tag{2.7}$$

For a sequence with convergence rate r, *asymptotic error constant*, γ is the limit above, i.e.

$$\gamma = \lim_{k\to\infty} \frac{\|x_{k+1} - x^*\|}{\|x_k - x^*\|^r} \tag{2.8}$$

To understand this better, lets assume that we have ideal convergence behavior so that condition (2.8) is true for every iteration $k+1$ and we do not have to take the limit. Then we can write

$$\|x_{k+1} - x^*\| = \gamma \|x_k - x^*\|^r \quad \text{for all } k. \tag{2.9}$$

The larger r is, the faster the convergence. When $r = 1$ we have *linear convergence*, and $\|x_{k+1} - x^*\| = \gamma\|x_k - x^*\|$. If $0 < \gamma < 1$, then the norm of the error decreases by a constant factor for every iteration. If $\gamma > 1$, the sequence diverges. When we have linear convergence, the number of iterations for a given level of convergence varies widely depending on the value of the asymptotic error constant.

If $\gamma = 0$ when $r = 1$, we have a special case: *superlinear convergence*. If $r = 1$ and $\gamma = 1$, we have *sublinear convergence*.

When $r = 2$, we have *quadratic convergence*. This is highly desirable, since the convergence is rapid and independent of the asymptotic error constant. For example, if $\gamma = 1$ and the initial error is $\|x_0 - x^*\| = 10^{-1}$, then the sequence of errors will be $10^{-1}, 10^{-2}, 10^{-4}, 10^{-8}, 10^{-16}$, i.e., the number of correct digits doubles for every iteration, and you can achieve double precision in four iterations!

For n dimensions, x is a vector with n components and we have to rethink the definition of the error. We could use, for example, $||x_k - x^*||$. However, this depends on the scaling of x, so we should normalize with respect to the norm of the current vector, i.e.

$$\frac{||x_k - x^*||}{||x_k||} \tag{2.10}$$

One potential problem with this definition is that x_k might be zero. To fix this, we write,

$$\frac{||x_k - x^*||}{1 + ||x_k||} \tag{2.11}$$

Another potential problem arises from situations where the gradients are large, i.e., even if we have a small $||x_k - x^*||$, $|f(x_k) - f(x^*)|$ is large. Thus, we should use a combined quantity, such as,

$$\frac{||x_k - x^*||}{1 + ||x_k||} + \frac{|f(x_k) - f(x^*)|}{1 + |f(x_k)|} \tag{2.12}$$

One final issue is that when we perform numerical optimization, x^* is usually not known! However, you can monitor the progress of your algorithm using the difference between successive steps,

$$\frac{||x_{k+1} - x_k||}{1 + ||x_k||} + \frac{|f(x_{k+1}) - f(x_k)|}{1 + |f(x_k)|} \tag{2.13}$$

Sometimes, you might just use the second fraction in the above term. For example, if f^* is known and nonzero then

$$\frac{|f(x_k) - f^*|}{|f^*|} \tag{2.14}$$

is a useful metric.

For gradient-based methods, perhaps the most commonly used convergence metric is simply the norm of the gradient:

$$\|g\| \tag{2.15}$$

Recall that the first-order optimality condition in 1D is $f'(x) = 0$. As we will see in Chapter 4, the n-dimensional analog is $\|g\| = 0$, so the gradient norm provides a natural convergence criteria.

In practice, usually multiple convergence criteria are monitored. Convergence plots should use the iteration number k on the x-axis with linear scaling and the convergence metric on the y-axis with *log scaling*.

2.4 Root Finding

Solving the first-order optimality conditions, that is, finding x^* such that $f'(x^*) = 0$, is equivalent to finding the roots of the first derivative of the function to be minimized. Therefore, root finding methods can be used to find stationary points and are useful in function minimization.

Using machine precision, it is not possible find the exact zero, so we will be satisfied with finding an x^* that belongs to an interval $[a, b]$ such that the function g satisfies

$$g(a)g(b) < 0 \quad \text{and} \quad |a - b| < \varepsilon \tag{2.16}$$

where ε is a "small" tolerance. This tolerance might be dictated by the machine representation (using double precision this is usually 1×10^{-16}), the precision of the function evaluation, or a limit on the number of iterations we want to perform with the root finding algorithm.

2.4.1 Method of Bisection

Bisection is a *bracketing method*. This class of methods generate a set of nested intervals and requires an initial interval that contains the solution.

In this method for finding the zero of a function f, we first establish a bracket $[x_1, x_2]$ for which the function values $f_1 = f(x_1)$ and $f_2 = f(x_2)$ have opposite sign. The function is then evaluated at the midpoint, $x = \frac{1}{2}(x_1 + x_2)$. If $f(x)$ and f_1 have opposite signs, then x and f become x_2 and f_2 respectively. On the other hand, if $f(x)$ and f_1 have the same sign, then x and f become x_1 and f_1. This process is then repeated until the desired accuracy is obtained.

For an initial interval $[x_1, x_2]$, bisection yields the following interval at iteration k,

$$\delta_k = \frac{x_1 - x_2}{2^k} \tag{2.17}$$

Thus, to achieve a specified tolerance δ, we need $\log_2(x_1 - x_2)/\delta$ evaluations.

Bisection yields the smallest interval of uncertainty for a specified number of function evaluations, which is the best you can do for a first order root finding method. From the definition of rate of convergence, for $r = 1$,

$$\lim_{k\to\infty} = \frac{\delta_{k+1}}{\delta_k} = \frac{1}{2} \tag{2.18}$$

Thus this method converges linearly with asymptotic error constant $\gamma = 1/2$. To find the minimum of a function using bisection, we would evaluate the derivative of f at each iteration, instead of the value.

2.4.2 Newton's Method for Root Finding

Newton's method for finding a zero can be derived from the Taylor's series expansion about the current iteration x_k,

$$f(x_{k+1}) = f(x_k) + (x_{k+1} - x_k)f'(x_k) + \mathcal{O}((x_{k+1} - x_k)^2) \tag{2.19}$$

Ignoring the terms higher than order two and assuming the function's next iteration to be the root (i.e., $f(x_{k+1}) = 0$), we obtain,

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}. \tag{2.20}$$

This iterative procedure converges quadratically, so

$$\lim_{k\to\infty} \frac{|x_{k+1} - x^*|}{|x_k - x^*|^2} = \text{const.} \tag{2.21}$$

A pictoral representation of two Newton iterations is shown in Fig. 2.2.

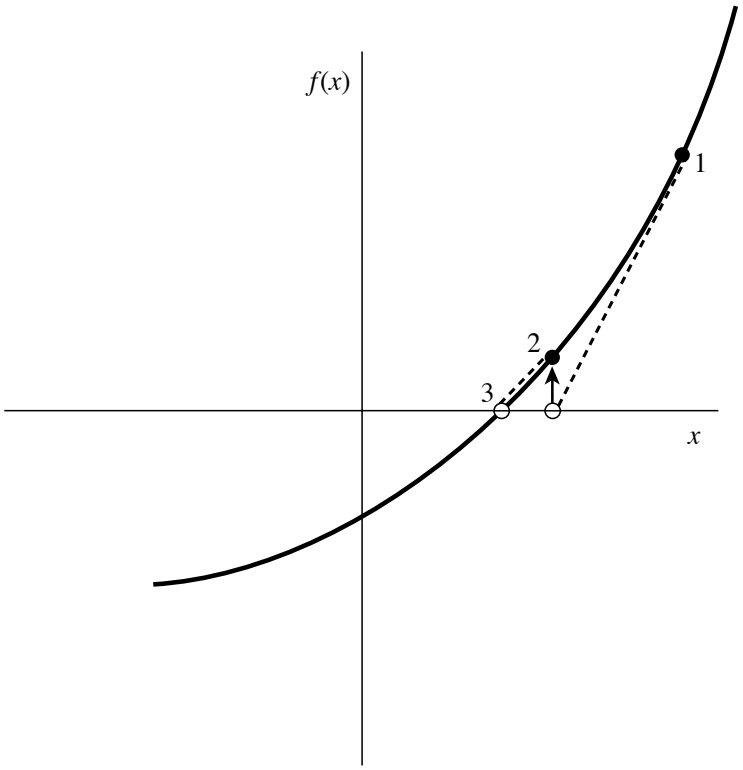


Figure 2.2: Iterations of Newton method for root finding

While having quadratic converge is a desirable property, Newton's method is not guaranteed to converge, and only works under certain conditions. Two examples where this method fails are shown in Fig. 2.3.

To minimize a function using Newton's method, we simply substitute the function for its first derivative and the first derivative by the second derivative,

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}. \tag{2.22}$$

Example 2.1. Function Minimization Using Newton's Method

Consider the following single-variable optimization problem

$$\begin{aligned} &\text{minimize} f(x) = (x-3)x^3(x-6)^4 \\ &\text{w.r.t.} x \end{aligned}$$

We solve this using Newton's method, and the results are shown in Fig. **??**

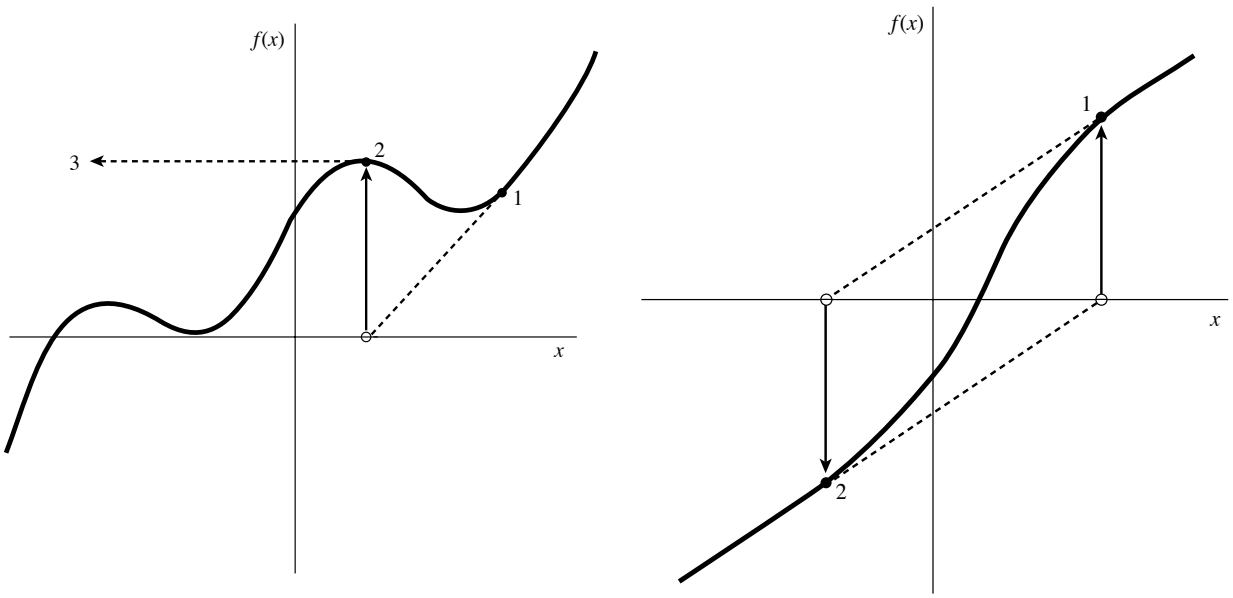


Figure 2.3: Cases where the Newton method fails

2.4.3 Secant Method

Newton's method requires the first derivative for each iteration (and the second derivative when applied to minimization). In some practical applications, it might not be possible to obtain this derivative analytically or it might just be troublesome.

If we use a forward-difference approximation for $f'(x_k)$ in Newton's method we obtain

$$x_{k+1} = x_k - f(x_k)\left(\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}\right). \tag{2.23}$$

which is the secant method (also known as "the poor-man's Newton method"). Under favorable conditions, this method has superlinear convergence ($1 < r < 2$), with $r \approx 1.6180$.

2.4.4 Brent's Method

Bracketing methods are robust but slow (linear convergence), whereas polynomial methods are fast (quadratic convergence) but susceptible to convergence failure. Brent's method combines the two approaches using bisection, a secant method, and quadratic interpolation and is widely used for 1-dimensional root finding. Brent's method demonstrates the same provable convergence behavior as bracketing methods, but converges faster (superlinearly).

The implementation is more complex and will not be enumerated here, but details can be found in separate references [2] and various available implementations.

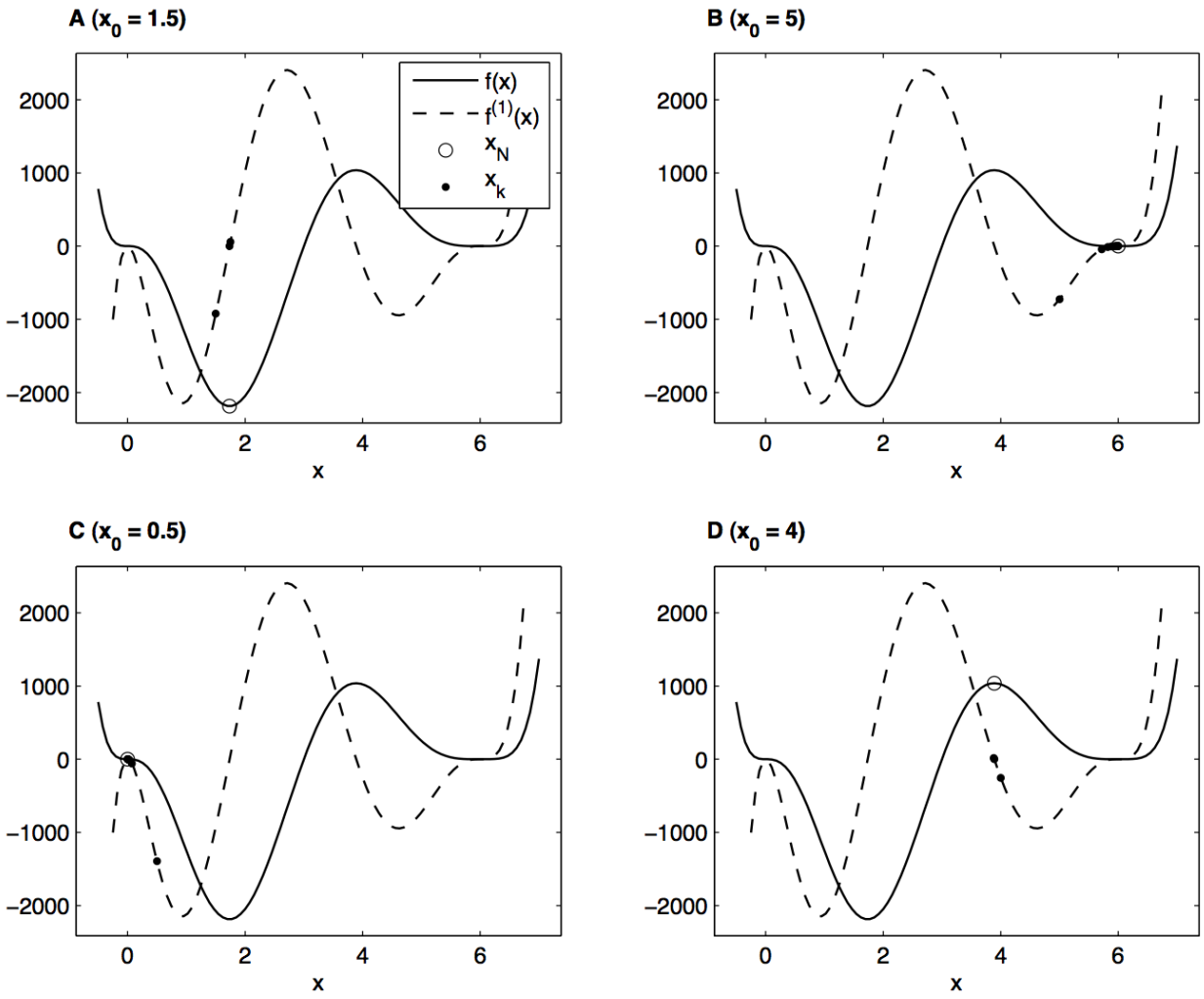


Figure 2.4: Newton's method with several different initial guesses. The x_k are Newton iterates, x_N is the converged solution, and $f^{(1)}(x)$ denotes the first derivative of $f(x)$.

2.5 Minimization Methods

A second class of 1D optimization methods is based directly on minimization rather than root finding. They parallel the methods discussed in root finding in that there are bracket methods, methods that make use of gradients, and hybrid approaches.

2.5.1 Golden Section Search

The golden section method is a function minimization method, as opposed to a root finding algorithm. It is analogous to the bisection algorithm in that it starts with an interval that is assumed to contain the minimum and then reduces that interval.

Say we start the search with an uncertainty interval $[0, 1]$. Unlike root finding methods, one function evaluation in the interval does not provide sufficient information to subdivide the interval; we need two function evaluations to determine which contains a minimum. We do not want to bias towards one side, so we choose the points symmetrically, say $1-\tau$ and τ, where $0 < \tau < 1$. In other words this produces two intervals both of length τ (Fig. 2.5). Like bisection, it would be inefficient to have one candidate interval smaller than the other. If the original interval is of length I_1 then this requirement produces two intervals $I_2 = \tau I_1$.

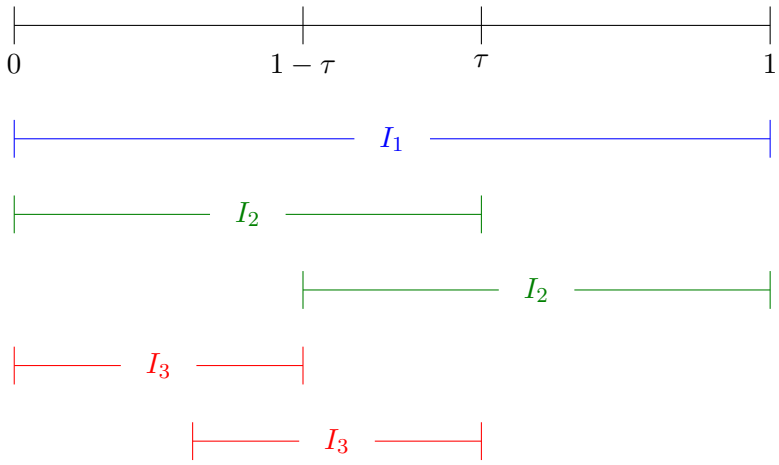


Figure 2.5: Golden Section Search interval division procedure.

Our second objective is to be able to reuse points. If we have to evaluate two new points every iteration then our method will be inefficient. In other words (referencing Fig. 2.5), if the left interval of I_2 contains a minimum and we apply the same division procedure, we would like it to automatically select $1-\tau$ as one of the interval points so that we could reuse its function evaluation. This implies that $I_1 = I_2 + I_3$. Substituting in the expressions $I_2 = \tau I_1$ and $I_3 = \tau I_2 = \tau^2 I_1$ yields

$$\tau^2 + \tau - 1 = 0 \tag{2.24}$$

We could also come to this same conclusion by looking at at the ratio of interval divisions such that they are equal.

$$\frac{\tau}{1} = \frac{1-\tau}{\tau} \Rightarrow \tau^2 + \tau - 1 = 0 \tag{2.25}$$

The positive solution of this equation is the golden ratio,

$$\tau = \frac{\sqrt{5}-1}{2} = 0.618033988749895\ldots \tag{2.26}$$

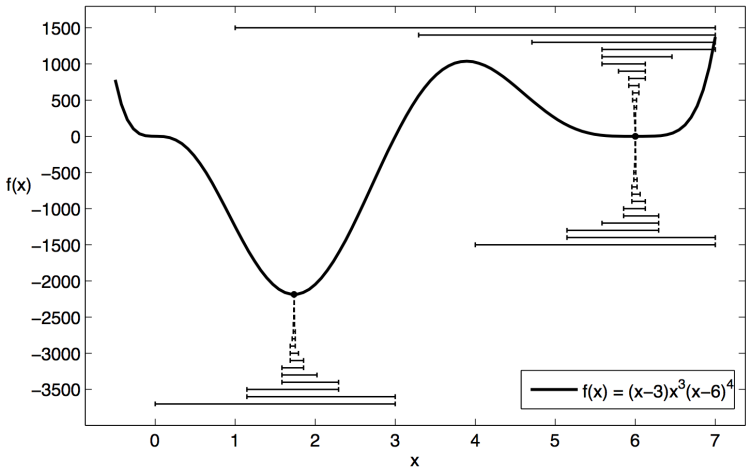


Figure 2.6: The golden section method with initial intervals $[0, 3]$, $[4, 7]$, and $[1, 7]$. The horizontal lines represent the sequences of the intervals of uncertainty.

(actually this is the inverse of the golden ratio as it is typically defined).

So we evaluate the function at $1 - \tau$ and τ, and then the two possible intervals are $[0, \tau]$ and $[1 - \tau, 1]$, which have the same size. If, say $[0, \tau]$ is selected, then the next two interior points would be $\tau(1 - \tau)$ and $\tau\tau$. But $\tau^2 = 1 - \tau$ from the definition (2.25), and we already have this point!

The Golden section method exhibits linear convergence.

Example 2.2. Line Search Using Golden Section

Consider the following single-variable optimization problem

$$\begin{aligned} \text{minimize} \quad & f(x) = (x-3)x^3(x-6)^4 \\ \text{w.r.t.} \quad & x \end{aligned}$$

Solve this using the golden section method. The result are shown in Fig. 2.6. Note the convergence to different optima, depending on the starting interval, and the fact that it might not converge to the best optimum within the starting interval.

2.5.2 Polynomial Interpolation

More efficient procedures use information about f gathered during iteration. One way of using this information is to produce an estimate of the function which we can easily minimize. The lowest order function that we can use for this purpose is a quadratic, since a linear function does not have a minimum.

Suppose we approximate f by

$$\tilde{f} = \frac{1}{2}ax^2 + bx + c. \tag{2.27}$$

If $a > 0$, the minimum of this function is $x^* = -b/a$.

To generate a quadratic approximation, three independent pieces of information are needed. For example, if we have the value of the function, its first derivative, and its second derivative at point x_k, we can write a quadratic approximation of the function value at x as the first three terms of a Taylor series

$$\tilde{f}(x) = f(x_k) + f'(x_k)(x - x_k) + \frac{1}{2}f''(x_k)(x - x_k)^2 \tag{2.28}$$

If $f''(x_k)$ is not zero, and setting $x = x_{k+1}$ this yields

$$x^* = -b/a \tag{2.29}$$

$$\Rightarrow x_{k+1} - x_k = -\frac{f'(x_k)}{f''(x_k)} \tag{2.30}$$

$$\Rightarrow x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)} \tag{2.31}$$

which is identical to Newton's method used to find a zero of the first derivative.

In general, providing an accurate second derivative is difficult. Instead, we can create a quadratic fit from three function values, or two function values and a gradient. Or, if derivatives are readily available then at x_k and x_{k+1} we will have four pieces of information (two function values and two gradients). In this case a cubic polynomial is a natural fit. Polynomials of degree larger than three are rarely used as they tend to create large oscillations. These alternative approaches will differ from Newton's method, but like Newton's method can lead to faster convergence.

2.5.3 Brent's Method

Like the Brent's method for root finding, there is a Brent's method for minimization with similar advantages. This method combines quadratic interpolation with Golden section search. Instead of using a function value and the first and second gradient to create a quadratic fit, Brent's method uses function evaluations at three different points. A new iteration point is estimated based on the polynomial fit. If the point falls outside of the bounding interval or requires a move that is too large, then the polynomial fit is rejected and the algorithms falls back to Golden section search for that iteration.

2.6 Line Search Techniques

Refer again to Fig. 2.1 and recall that our actual goal is not 1D optimization, but to perform a line search along a given direction p_k. There are two important differences between a line search and the 1D optimization methods discussed so far.

1. A line search is one-dimensional but occurs along a line in n-dimensional space. The previous algorithms were written along the x-axis, but must be generalized for a general line search.

2. A line search is generally not interested in finding the exact optimum, but rather a point that is "good enough" quickly. This criteria is called the Wolfe conditions and will be discussed in the next section.

The one-dimensional methods discussed previously can be generalized in a straightforward manner. First, recall the concept of a gradient and a directional derivative from multivariable calculus. The gradient g is a vector of partial derivatives in each dimension:

$$g(x) \equiv \nabla f(x) \equiv \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} \tag{2.32}$$

The directional derivative is the derivative in a given direction. The derivative of the function f in the direction p is given by

$$\frac{df}{dp} = g^T \hat{p} \tag{2.33}$$

where $\hat{p}$ is a unit vector in the p direction. Thus, in the above algorithms we would replace $f'(x)$ with $g^T\hat{p}$.

Line search methods are related to single-variable optimization methods, as they address the problem of minimizing a multivariable function along a line, which is a subproblem in many gradient-based optimization method. After a gradient-based optimizer has computed a search direction p_k, it must decide how far to move along that direction. The step can be written as

$$x_{k+1} = x_k + \alpha_k p_k \tag{2.34}$$

where the positive scalar α_k is the *step length*.

Most algorithms require that p_k be a *descent direction*, i.e., that $g_k^T p_k < 0$, since this guarantees that f can be reduced by stepping some distance along this direction. The question then becomes, how far in that direction we should move.

We want to compute a step length α_k that yields a substantial reduction in f, but we do not want to spend too much computational effort in making the choice. Ideally, we would find the global minimum of $f(x_k + \alpha_k p_k)$ with respect to α_k but in general, it is too expensive to compute this value. Even to find a local minimizer usually requires too many evaluations of the objective function f and possibly its gradient g. More practical methods perform an *inexact* line search that achieves adequate reductions of f at reasonable cost.

2.6.1 Wolfe Conditions

A typical line search involves trying a sequence of step lengths, accepting the first that satisfies certain conditions. A common condition requires that α_k should yield a *sufficient decrease* of f, as given by the inequality

$$f(x_k + \alpha p_k) \leq f(x_k) + \mu_1 \alpha g_k^T p_k \tag{2.35}$$

for a constant $0 \leq \mu_1 \leq 1$. In practice, this constant is small, say $\mu_1 = 10^{-4}$. This sufficient decrease condition is also known as Armijo's rule.

Any sufficiently small step can satisfy the sufficient decrease condition, so in order to prevent steps that are too small we need a second requirement called the *curvature condition*, which can be stated as

$$g(x_k + \alpha p_k)^T p_k \geq \mu_2 g_k^T p_k \tag{2.36}$$

where $\mu_1 \leq \mu_2 \leq 1$, and $g(x_k + \alpha p_k)^T p_k$ is the derivative of $f(x_k + \alpha p_k)$ with respect to α_k. This condition requires that the slope of the univariate function at the new point be greater. Since we start with a negative slope, the gradient at the new point must be either less negative or positive. The intuition is that if the gradient is even more negative then we should be taking a larger step size to take advantage of the expected decrease (in other words the step size is too small). Typical values of μ_2 are 0.9 when using a Newton type method and 0.1 when a conjugate gradient methods is used. The sufficient decrease and curvature conditions are known collectively as the *Wolfe conditions*.

We can also modify the curvature condition to force α_k to lie in a broad neighborhood of a local minimizer or stationary point and obtain the *strong Wolfe conditions*

$$f(x_k + \alpha p_k) \leq f(x_k) + \mu_1 \alpha g_k^T p_k. \tag{2.37}$$

$$\left| g(x_k + \alpha p_k)^T p_k \right| \leq \mu_2 \left| g_k^T p_k \right|, \tag{2.38}$$

where $0 < \mu_1 < \mu_2 < 1$. The only difference when comparing with the Wolfe conditions is that we do not allow points where the derivative has a positive value that is too large and therefore exclude points that are far from the stationary points. Figure 2.7 graphically demonstrates acceptable intervals that satisfy the Wolfe conditions.

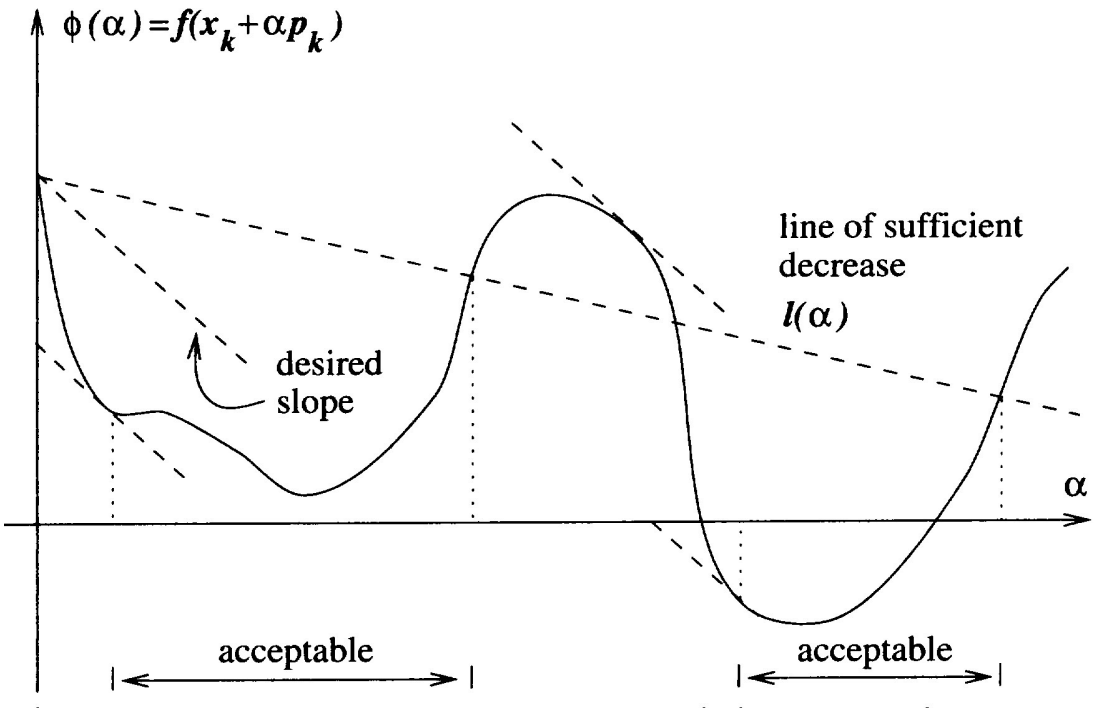


Figure 2.7: Acceptable steps for the strong Wolfe conditions.

2.6.2 Sufficient Decrease and Backtracking

An important concept in a line search is that of backtracking. Most of the optimization methods will not only provide a search direction p_k, but also an initial estimate of the step size α. If the function value at the suggested step size turns out to be poor, then we are too far away for our model to be reliable. That means we should backtrack towards the region where the model will be better. Because our search direction is a descent direction we know that if we backtrack enough we will achieve a decrease in function value.

The illustrate, Algorithm 1 shows a *simple backtracking line search*. This algorithm consists of guessing a maximum step, then successively decreasing that step until the new point satisfies the sufficient decrease condition.

Algorithm 1 Backtracking line search algorithm

Input: $\alpha > 0$ and $0 < \rho < 1$
Output: α_k
1: **repeat**
2: $\quad \alpha \leftarrow \rho\alpha$
3: **until** $f(x_k + \alpha p_k) \leq f(x_k) + \mu_1 \alpha g_k^T p_k$
4: $\alpha_k \leftarrow \alpha$

This method is very simplistic, and its main purpose is just to illustrate the concept of backtracking. Real line search methods combine the concepts of backtracking, bracketing, and 1D optimization methods until the Wolfe conditions are satisfied. The next section gives an example of one such line search, but it is just one example and many variants of higher sophistication exist.

2.6.3 Line Search Algorithm Satisfying the Strong Wolfe Conditions

To simplify the notation, we define the univariate function

$$\phi(\alpha) = f(x_k + \alpha p_k) \tag{2.39}$$

Then at the starting point of the line search, $\phi(0) = f(x_k)$. The slope of the univariate function, is the projection of the n-dimensional gradient onto the search direction p_k, i.e.,

$$\phi'(\alpha) = g(x_k + \alpha p_k)^T p_k \tag{2.40}$$

This procedure has two stages:

1. Begins with trial α_1, and keeps increasing it until it finds either an acceptable step length or an interval that brackets the desired step lengths.

2. In the latter case, a second stage (the `zoom` algorithm below) is performed that decreases the size of the interval until an acceptable step length is found.

The first stage of this line search is detailed in Algorithm 2.

The algorithm for the second stage, the `zoom`$(\alpha_{\text{low}}, \alpha_{\text{high}})$ function is given in Algorithm 3. To find the trial point, we can use cubic or quadratic interpolation to find an estimate of the minimum within the interval. This results in much better convergence that if we use the golden search, for example.

Note that sequence $\{\alpha_i\}$ is monotonically increasing, but that the order of the arguments supplied to `zoom` may vary. The order of inputs for the call is `zoom`$(\alpha_{\text{low}}, \alpha_{\text{high}})$, where:

1. the interval between α_{low}, and α_{high} contains step lengths that satisfy the strong Wolfe conditions

2. α_{low} is the one giving the lowest function value

3. α_{high} is chosen such that $\phi'(\alpha_{\text{low}})(\alpha_{\text{high}} - \alpha_{\text{low}}) < 0$

The interpolation that determines α_j should be safeguarded, to ensure that it is not too close to the endpoints of the interval.

Using an algorithm based on the strong Wolfe conditions (as opposed to the plain Wolfe conditions) has the advantage that by decreasing μ_2, we can force α to be arbitrarily close to the local minimum.

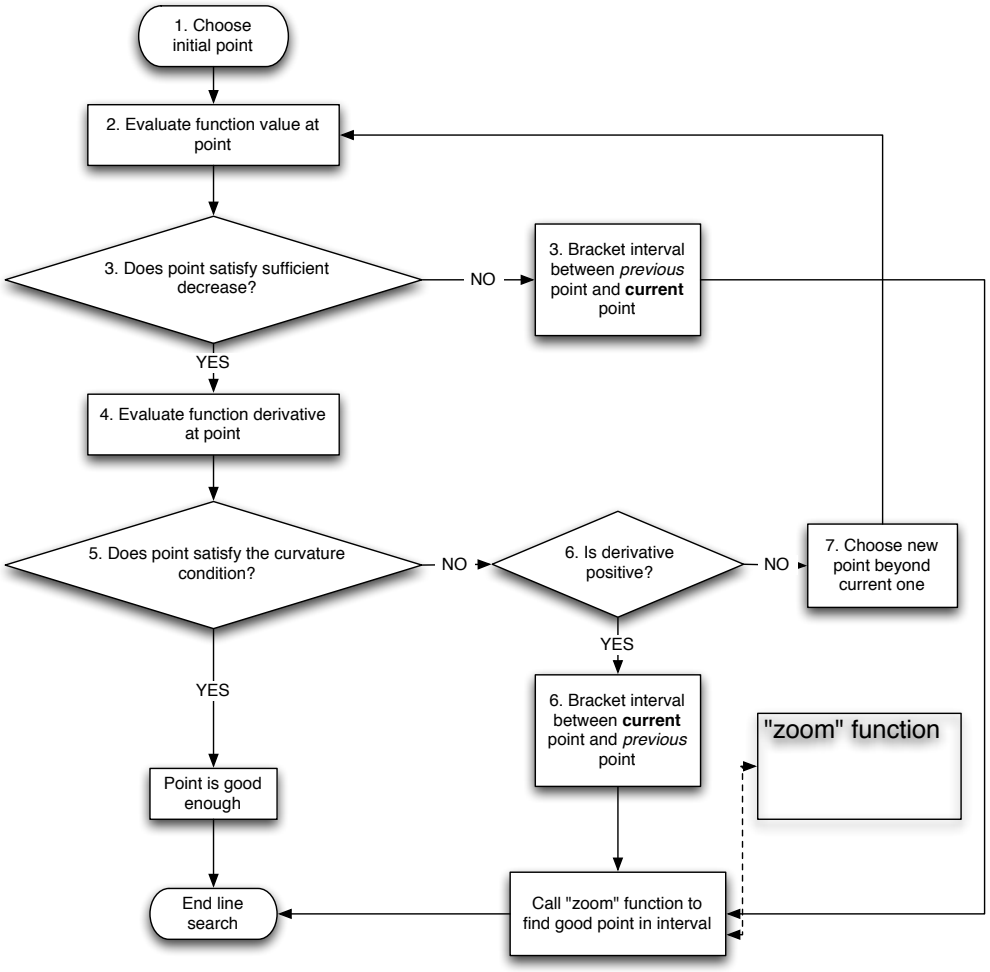


Figure 2.8: Line search algorithm satisfying the strong Wolfe conditions

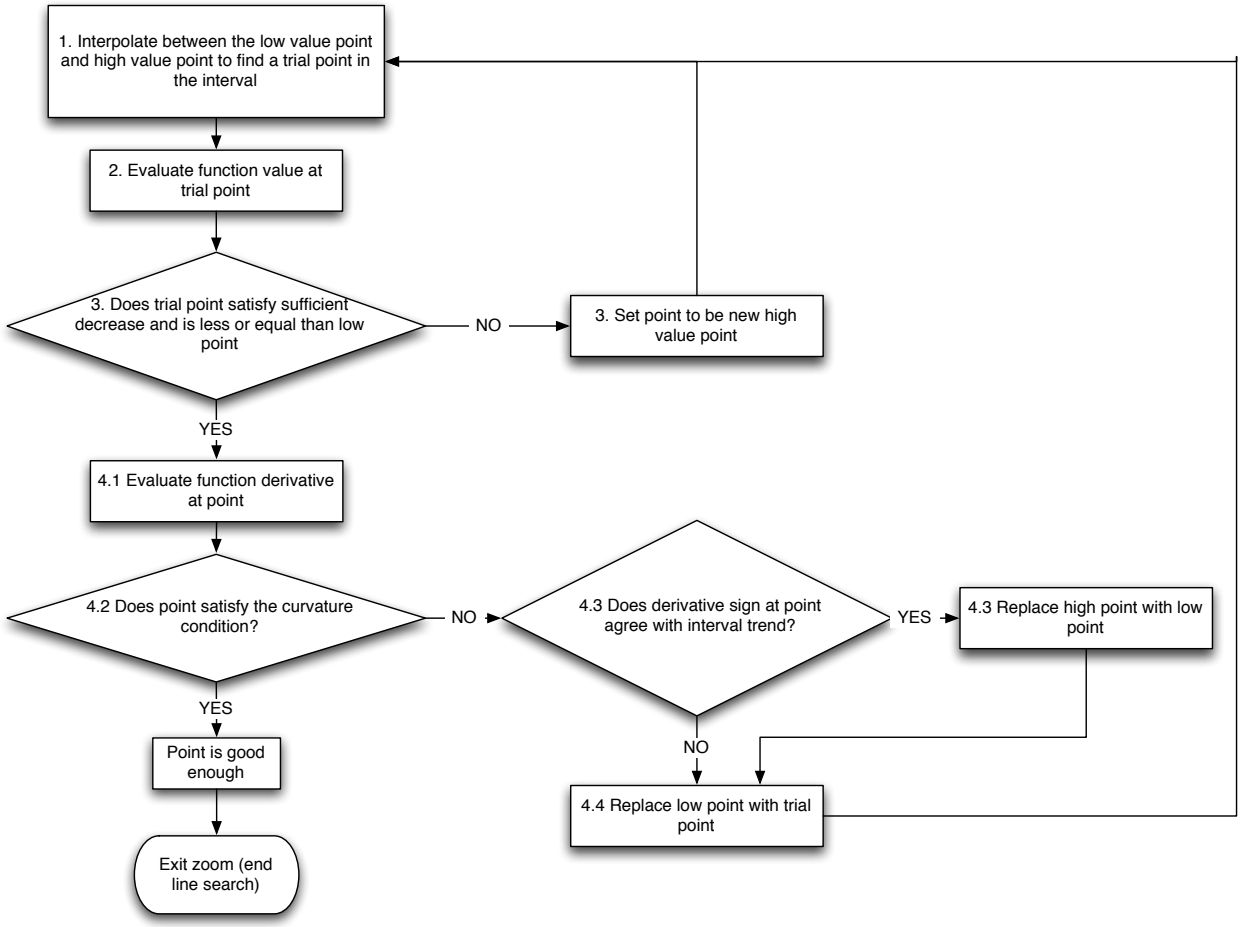


Figure 2.9: "Zoom" function in the line search algorithm satisfying the strong Wolfe conditions

Algorithm 2 Line search algorithm

Input: $\alpha_1 > 0$ and $\alpha_{\max}$
Output: α_*

$\alpha_0 = 0$
$i \leftarrow 1$
repeat
  Evaluate $\phi(\alpha_i)$
  if $[\phi(\alpha_i) > \phi(0) + \mu_1 \alpha_i \phi'(0)]$ or $[\phi(\alpha_i) > \phi(\alpha_{i-1})$ and $i > 1]$ **then**
    $\alpha_* \leftarrow \texttt{zoom}(\alpha_{i-1}, \alpha_i)$
    return α_*
  end if
  Evaluate $\phi'(\alpha_i)$
  if $|\phi'(\alpha_i)| \leq -\mu_2 \phi'(0)$ **then**
    return $\alpha_* \leftarrow \alpha_i$
  else if $\phi'(\alpha_i) \geq 0$ **then**
    $\alpha_* \leftarrow \texttt{zoom}(\alpha_i, \alpha_{i-1})$
    return α_*
  else
    Choose α_{i+1} such that $\alpha_i < \alpha_{i+1} < \alpha_{\max}$
  end if
  $i \leftarrow i + 1$
until

Algorithm 3 Zoom function for the line search algorithm

Input: $\alpha_{\text{low}}, \alpha_{\text{high}}$
Output: α_*

$j \leftarrow 0$
repeat
  Find a trial point α_j between α_{low} and α_{high}
  Evaluate $\phi(\alpha_j)$
  if $\phi(\alpha_j) > \phi(0) + \mu_1 \alpha_j \phi'(0)$ or $\phi(\alpha_j) > \phi(\alpha_{\text{low}})$ **then**
    $\alpha_{\text{high}} \leftarrow \alpha_j$
  else
    Evaluate $\phi'(\alpha_j)$
    if $|\phi'(\alpha_j)| \leq -\mu_2 \phi'(0)$ **then**
      $\alpha_* = \alpha_j$
      return α_*
    else if $\phi'(\alpha_j)(\alpha_{\text{high}} - \alpha_{\text{low}}) \geq 0$ **then**
      $\alpha_{\text{high}} \leftarrow \alpha_{\text{low}}$
    end if
    $\alpha_{\text{low}} \leftarrow \alpha_j$
  end if
  $j \leftarrow j + 1$
until

Example 2.3. Line Search Algorithm Using Strong Wolfe Conditions

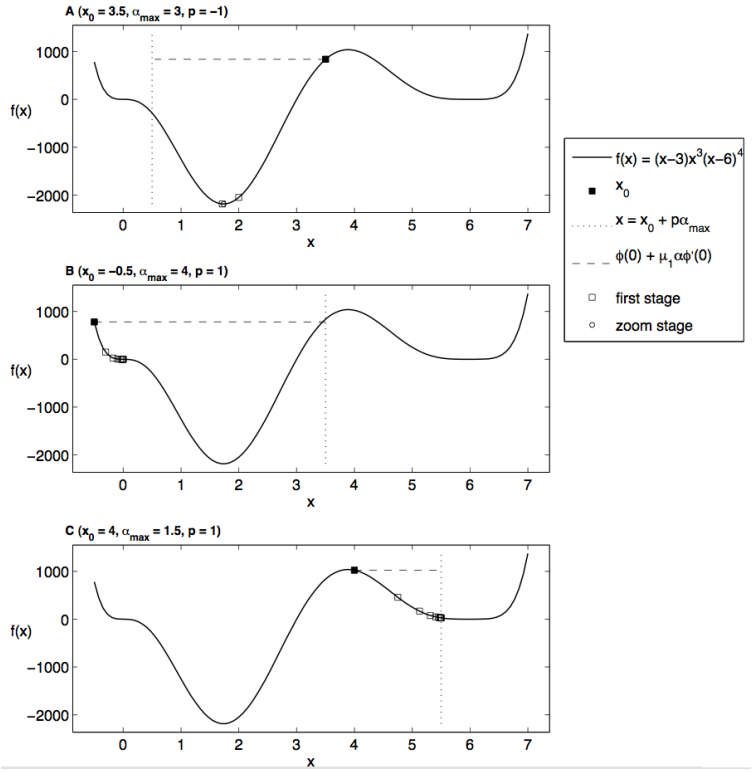


Figure 2.10: The line search algorithm iterations. The first stage is marked with square labels and the zoom stage is marked with circles.

Bibliography

[1] Ashok D. Belegundu and Tirupathi R. Chandrupatla. *Optimization Concepts and Applications in Engineering*, chapter 2. Prentice Hall, 1999.

[2] Richard P. Brent. *Algorithms for Minimization Without Derivatives*. Courier Corporation, Jun 2013. ISBN 0486143686.

[3] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization*, chapter 4. Academic Press, 1981.

[4] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*, chapter 3. Springer-Verlag, 1999.

Chapter 3

Gradient-Based Optimization

3.1 Introduction

In Chapter 2 we described methods to minimize (or at least decrease) a function of one variable. While problems with one variable do exist in MDO, most problems of interest involve multiple design variables. In this chapter we consider methods to solve such problems, restricting ourselves to smooth unconstrained problems. Later, we extend this to constrained problems in Chapter 5, and remove the smoothness requirement in Chapter 6.

The unconstrained optimization problem can be stated as,

$$\begin{aligned} &\text{minimize} && f(x) \\ &\text{with respect to} && x \in \mathbb{R}^n \end{aligned} \tag{3.1}$$

where x is the n-vector $x = [x_1, x_2, \ldots, x_n]^T$. The objective function f can be nonlinear, but one important assumption in this chapter is that f is a sufficiently smooth function of x: in some cases we will demand that f has continuous first (partial) derivatives, and in others we require f to also have continuous second (partial) derivatives.

For large numbers of variables n, gradient-based methods are usually the most efficient algorithms. This class of methods uses the gradient of the objective function to determine the most promising directions along which we should search. And here is where the line search comes in: it provides a way to search for a better point along a line in n-dimensional space.

Most algorithms for unconstrained gradient-based optimization can be described as shown in Algorithm 14. The outer loop represents the *major iterations*. The design variables are updated at each major iteration k using

$$x_{k+1} = x_k + \underbrace{\alpha_k p_k}_{\Delta x_k} \tag{3.2}$$

where p_k is the search direction for major iteration k, and α_k is the accepted step length from the line search. The line search usually involves multiple iterations that do not count towards the major iterations. This is an important distinction that needs to be considered when comparing the computational cost of the various approaches.

The two iterative loops represent the two subproblems in this type of algorithm: 1) The computation a search direction p_k, and; 2) The search for an acceptable step size (controlled by α_k). These two subproblems are illustrated in Fig. 3.1. The various types of gradient-based algorithms are classified based on the method that is used for computing the search direction (the first subproblem). Any line search that satisfies sufficient decrease can be used with a given gradient-based method.

Algorithm 4 General algorithm for smooth functions

Input: Initial guess, x_0
Output: Optimum, x^*
  $k \leftarrow 0$
  while Not converged **do**
    Compute a search direction p_k
    Find a step length α_k, such that $f(x_k + \alpha_k p_k) < f(x_k)$ (the curvature condition may also be included)
    Update the design variables: $x_{k+1} \leftarrow x_k + \alpha_k p_k$
    $k \leftarrow k + 1$
  end while

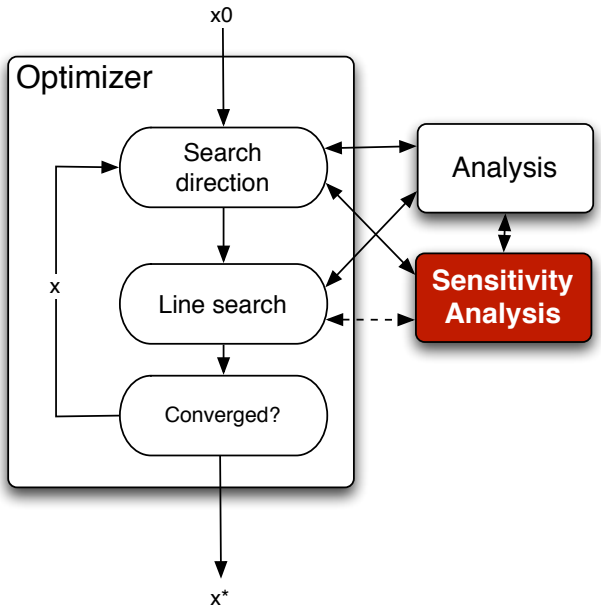


Figure 3.1: General gradient-based method and its relation to sensitivity analysis. "Analysis" is the evaluation of the objective function f, and "Sensitivity Analysis" the evaluation of ∇f

3.2 Gradients and Hessians

Consider a function $f(x)$. The *gradient* of this function is given by the vector of partial derivatives with respect to each of the independent variables,

$$\nabla f(x) \equiv g(x) \equiv \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} \tag{3.3}$$

In the multivariate case, the gradient vector is perpendicular to the the *hyperplane* tangent to the contour surfaces of constant f.

Higher derivatives of multi-variable functions are defined as in the single-variable case, but note that the number of gradient components increase by a factor of n for each differentiation.

While the gradient of a function of n variables is an n-vector, the "second derivative" of an n-variable function is defined by n^2 partial derivatives (the derivatives of the n first partial derivatives with respect to the n variables):

$$\frac{\partial^2 f}{\partial x_i \partial x_j}, \quad i \neq j \quad \text{and} \quad \frac{\partial^2 f}{\partial x_i^2}, \quad i = j. \tag{3.4}$$

If the partial derivatives $\partial f/\partial x_i$, $\partial f/\partial x_j$ and $\partial^2 f/\partial x_i \partial x_j$ are continuous and f is single valued, $\partial^2 f/\partial x_i \partial x_j = \partial^2 f/\partial x_j \partial x_i$. Therefore the second-order partial derivatives can be represented by a square symmetric matrix called the *Hessian matrix*,

$$\nabla^2 f(x) \equiv H(x) \equiv \begin{bmatrix} \dfrac{\partial^2 f}{\partial^2 x_1} & \cdots & \dfrac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & & \vdots \\ \dfrac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \dfrac{\partial^2 f}{\partial^2 x_n} \end{bmatrix} \tag{3.5}$$

which contains $n(n+1)/2$ independent elements.

If f is quadratic, the Hessian of f is constant, and the function can be expressed as

$$f(x) = \frac{1}{2} x^T H x + g^T x + \alpha. \tag{3.6}$$

3.3 Optimality Conditions

As in the single-variable case, the optimality conditions can be derived from the Taylor-series expansion of f about x^*:

$$f(x^* + \varepsilon p) = f(x^*) + \varepsilon p^T g(x^*) + \frac{1}{2} \varepsilon^2 p^T H(x^* + \varepsilon \theta p) p, \tag{3.7}$$

where $\theta \in (0, 1)$, ε is a scalar, and p is an n-vector.

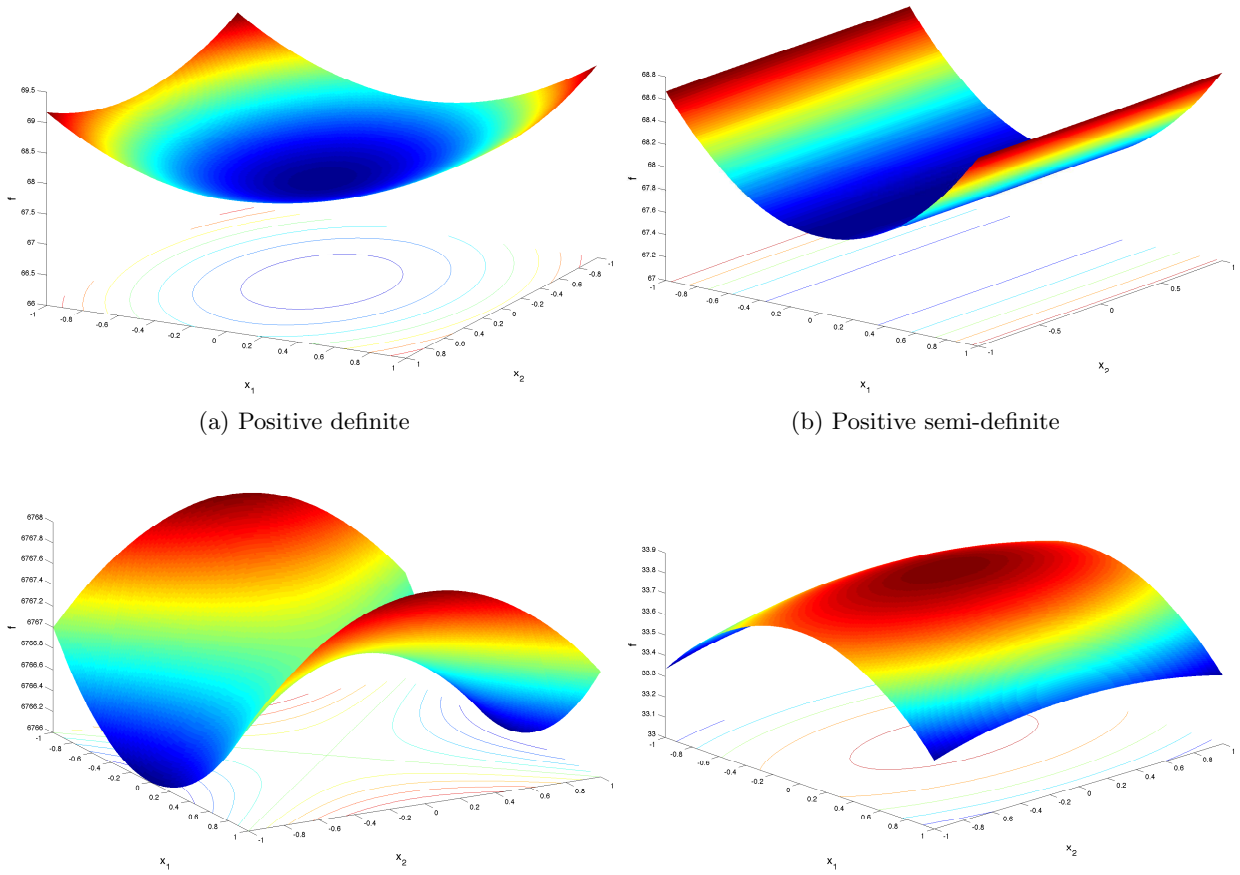
For x^* to be a local minimum, then for any vector p there must be a finite ε such that $f(x^* + \varepsilon p) \geq f(x^*)$, i.e., there is a neighborhood in which this condition holds. If this condition is satisfied, then $f(x^* + \varepsilon p) - f(x^*) \geq 0$ and the first and second order terms in the Taylor-series expansion must be greater than or equal to zero.

As in the single variable case, and for the same reason, we start by considering the first order terms. Since p is an arbitrary vector and ε can be positive or negative, every component of the gradient vector $g(x^*)$ must be zero.

Now we have to consider the second order term, $\varepsilon^2 p^T H(x^* + \varepsilon \theta p) p$. For this term to be non-negative, $H(x^* + \varepsilon \theta p)$ has to be positive semi-definite, and by continuity, the Hessian at the optimum, $H(x^*)$ must also be positive semi-definite.

- The matrix $H \in \mathbb{R}^{n \times n}$ is *positive definite* if $p^T H p > 0$ for all nonzero vectors $p \in \mathbb{R}^n$. If $H = H^T$ then all the eigenvalues of H are strictly positive.
- The matrix $H \in \mathbb{R}^{n \times n}$ is *positive semi-definite* if $p^T H p \geq 0$ for all vectors $p \in \mathbb{R}^n$. If $H = H^T$ then the eigenvalues of H are positive or zero.

- The matrix $H \in \mathbb{R}^{n\times n}$ is *indefinite* if there exists $p, q \in \mathbb{R}^n$ such that $p^T Hp > 0$ and $q^T Hq < 0$. If $H = H^T$ then H has eigenvalues of mixed sign.
- The matrix $H \in \mathbb{R}^{n\times n}$ is *negative definite* if $p^T Hp < 0$ for all nonzero vectors $p \in \mathbb{R}^n$. If $H = H^T$ then all the eigenvalues of H are strictly negative



(a) Positive definite

(b) Positive semi-definite

(c) Indefinite

(d) Negative definite

Figure 3.2: Various possibilities for a quadratic function

Necessary conditions (for a local minimum):

$$\|g(x^*)\| = 0 \quad \text{and} \quad H(x^*) \quad \text{is positive semi-definite.} \tag{3.8}$$

Sufficient conditions (for a strong local minimum):

$$\|g(x^*)\| = 0 \quad \text{and} \quad H(x^*) \quad \text{is positive definite.} \tag{3.9}$$

Example 3.4. Critical Points of a Function Consider the function:

$$f(x) = 1.5x_1^2 + x_2^2 - 2x_1x_2 + 2x_1^3 + 0.5x_1^4$$

Find all stationary points of f and classify them.

Solve $\nabla f(x) = 0$, get three solutions:

$$\begin{aligned} (0,0) &\quad \text{local minimum} \\ 1/2(-3-\sqrt{7}, -3-\sqrt{7}) &\quad \text{global minimum} \\ 1/2(-3+\sqrt{7}, -3+\sqrt{7}) &\quad \text{saddle point} \end{aligned}$$

To establish the type of point, we have to determine if the Hessian is positive definite and compare the values of the function at the points.

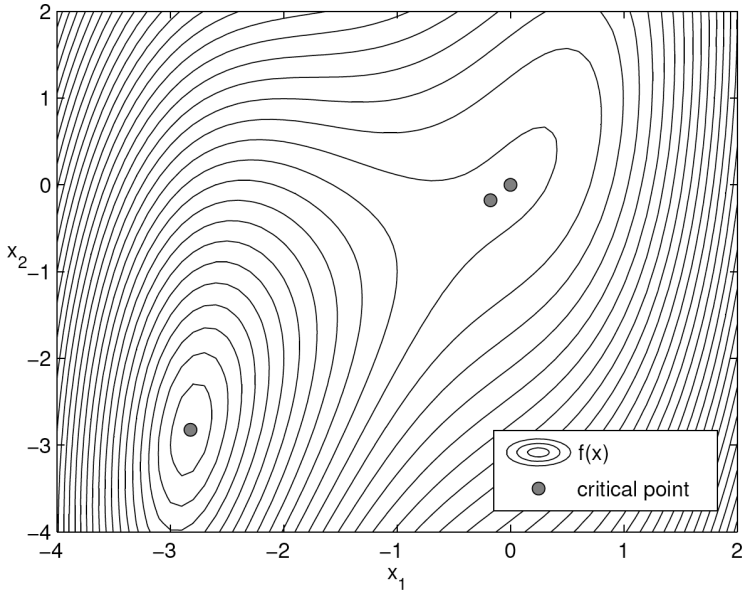


Figure 3.3: Critical points of $f(x) = 1.5x_1^2 + x_2^2 - 2x_1x_2 + 2x_1^3 + 0.5x_1^4$

3.4 Steepest Descent Method

The steepest descent method uses the gradient vector at x_k as the search direction for the major iteration k. As mentioned previously, the gradient vector is orthogonal to the plane tangent to the isosurfaces of the function. The gradient vector, $g(x_k)$, is also the direction of maximum rate of change (maximum increase) of the function at that point. This rate of change is given by the norm, $\|g(x_k)\|$.

Algorithm 5 Steepest descent

Input: Initial guess, x_0, convergence tolerances, $\varepsilon_g, \varepsilon_a$ and ε_r.
Output: Optimum, x^*

$k \leftarrow 0$
repeat
  Compute the gradient of the objective function, $g(x_k) \equiv \nabla f(x_k)$
  Compute the normalized search direction, $p_k \leftarrow -g(x_k)/\|g(x_k)\|$
  Perform line search to find step length α_k
  Update the current point, $x_{k+1} \leftarrow x_k + \alpha_k p_k$
  $k \leftarrow k + 1$
until $|f(x_k) - f(x_{k-1})| \leq \varepsilon_a + \varepsilon_r |f(x_{k-1})|$ and $\|g(x_{k-1})\| \leq \varepsilon_g$

Here, $|f(x_{k+1}) - f(x_k)| \leq \varepsilon_a + \varepsilon_r |f(x_k)|$ is a check for the successive reductions of f. ε_a is the absolute tolerance on the change in function value (usually small $\approx 10^{-6}$) and ε_r is the relative tolerance (usually set to 0.01).

If we use an exact line search, the steepest descent direction at each iteration is orthogonal to the previous one, i.e.,

$$\begin{aligned} & \frac{\mathrm{d}f(x_{k+1})}{\mathrm{d}\alpha} = 0 \\ \Rightarrow \quad & \frac{\partial f(x_{k+1})}{\partial x_{k+1}} \frac{\partial \left(x_k + \alpha p_k\right)}{\partial \alpha} = 0 \\ \Rightarrow \quad & \nabla^T f(x_{k+1}) p_k = 0 \\ \Rightarrow \quad & -g^T(x_{k+1}) g(x_k) = 0 \end{aligned}$$

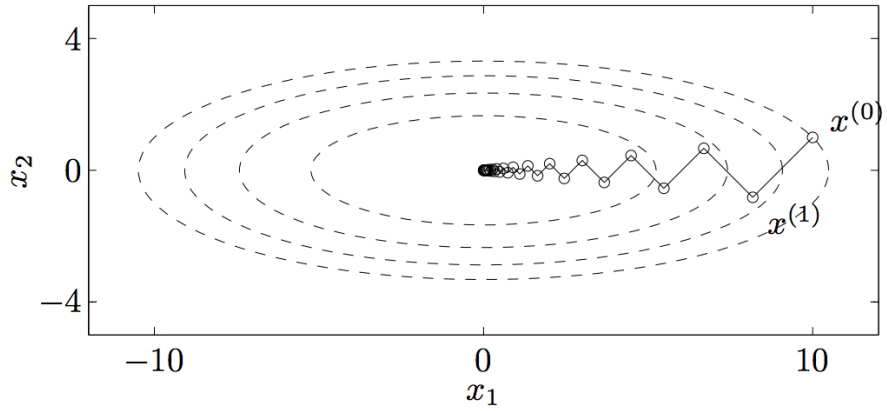
Because of this property of the search directions, the steepest descent method "zigzags" in the design space and is inefficient, in general. Although a substantial decrease may be observed in the first few iterations, the method is usually very slow to converge to a local optimum. In particular, while the algorithm is guaranteed to converge, it may take an infinite number of iterations. The rate of convergence is linear.

Consider, for example the quadratic function of two variables,

$$f(x) = (1/2)(x_1^2 + 10x_2^2)$$

.

The following figure shows the iterations given by steepest descent when started at $x_0 = (10, 1)$.



For steepest descent and other gradient methods that do not produce well-scaled search directions, we need to use other information to guess a step length. One strategy is to assume that the first-order change in x_k will be the same as the one obtained in the previous step. i.e, that $\bar{\alpha} g_k^T p_k = \alpha_{k-1} g_{k-1}^T p_{k-1}$ and therefore:

$$\bar{\alpha} = \alpha_{k-1} \frac{g_{k-1}^T p_{k-1}}{g_k^T p_k}. \tag{3.10}$$

Example 3.5. Steepest Descent Consider the following function.

$$f(x_1, x_2) = 1 - e^{-(10x_1^2 + x_2^2)}. \tag{3.11}$$

The function f is not quadratic, but, as $|x_1|$ and $|x_2| \to 0$, we see that

$$f(x_1, x_2 = 10x_1^2 + x_2^2 + \mathrm{O}(x_1^4) + \mathrm{O}(x_2^4)$$

Thus, this function is essentially a quadratic near the minimum $(0, 0)^T$. Fig. 3.4 shows the path taken by steepest descent starting from the initial point, $(-0.1, 0.6)^T$.

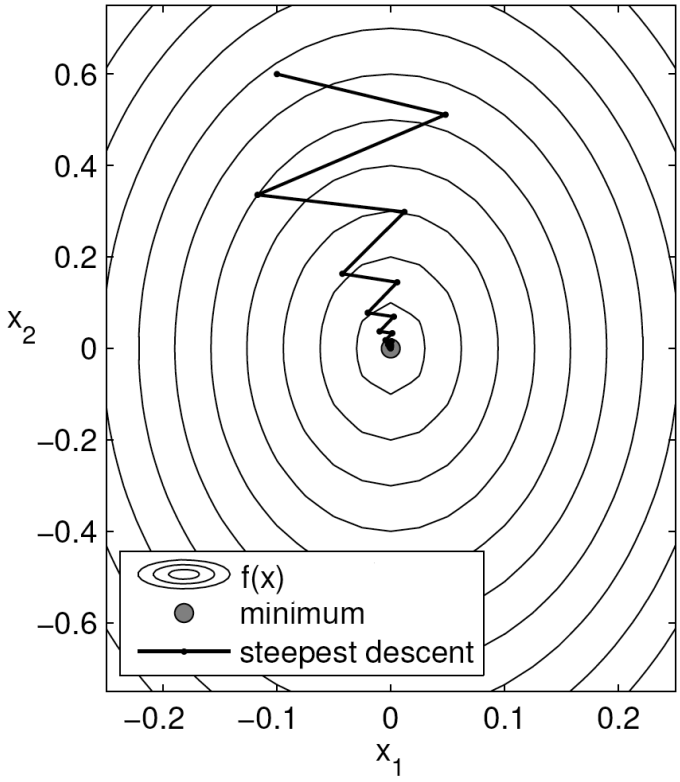


Figure 3.4: Solution path of the steepest descent method

3.5 Conjugate Gradient Method

A small modification to the steepest descent method takes into account the history of the gradients to move more directly towards the optimum.

Suppose we want to minimize a convex quadratic function

$$\phi(x) = \frac{1}{2}x^T Ax - b^T x \tag{3.12}$$

where A is an $n \times n$ matrix that is symmetric and positive definite. Differentiating this with respect to x we obtain,

$$\nabla\phi(x) = Ax - b \equiv r(x). \tag{3.13}$$

Minimizing the quadratic is thus equivalent to solving the linear system,

$$\nabla\phi = 0 \Rightarrow Ax = b. \tag{3.14}$$

The conjugate gradient method is an iterative method for solving linear systems of equations such as this one.

A set of nonzero vectors $\{p_0, p_1, \ldots, p_{n-1}\}$ is *conjugate* with respect to A if

$$p_i^T A p_j = 0, \quad \text{for all} \quad i \neq j. \tag{3.15}$$

Suppose that we start from a point x_0 and a set of conjugate directions $\{p_0, p_1, \ldots, p_{n-1}\}$ to generate a sequence $\{x_k\}$ where

$$x_{k+1} = x_k + \alpha_k p_k \tag{3.16}$$

where α_k is the minimizer of ϕ along $x_k + \alpha p_k$, given by

$$\begin{aligned} \frac{d\phi(x_k + \alpha p_k)}{d\alpha} = \frac{d}{d\alpha}\left[\frac{1}{2}(x_k + \alpha p_k)^T A(x_k + \alpha p_k) - b^T(x_k + \alpha p_k)\right] = 0 \\ \Rightarrow \quad p_k^T A(x_k + \alpha p_k) - p_k^T b = 0 \\ \Rightarrow p_k^T \underbrace{(Ax_k - b)}_{r_k} + \alpha p_k^T A p_k = 0 \end{aligned}$$

Rearranging the last equation we find

$$\alpha_k \equiv -\frac{r_k^T p_k}{p_k^T A p_k} \tag{3.17}$$

We will see that for any x_0 the sequence $\{x_k\}$ generated by the conjugate direction algorithm converges to the solution of the linear system in at most n steps.

The conjugate directions are linearly independent, so they span n-space. Therefore, we can expand the vector $x^* - x_0$ using the p_k as a basis (recall that x^* is the solution).

$$x^* - x_0 = \sigma_0 p_0 + \cdots + \sigma_{n-1} p_{n-1} \tag{3.18}$$

Premultiplying by $p_k^T A$ and using the conjugacy property we obtain

$$\sigma_k = \frac{p_k^T A(x^* - x_0)}{p_k^T A p_k} \tag{3.19}$$

Now we will show that the σ's are really the α's defined in (3.17). A given iteration point can be written as a function of the search directions and step sizes so far,

$$x_k = x_0 + \alpha_0 p_0 + \cdots + \alpha_{k-1} p_{k-1}. \tag{3.20}$$

Premultiplying by $p_k^T A$ and using the conjugacy property we obtain

$$p_k^T A(x_k - x_0) = 0. \tag{3.21}$$

Therefore

$$\begin{aligned} p_k^T A(x^* - x_0) &= p_k^T A(x^* - x_k + x_k - x_0) \\ &= p_k^T A(x^* - x_k) + \underbrace{p_k^T A(x_k - x_0)}_{=0} \\ &= p_k^T (\underbrace{Ax^*}_{=b} - Ax_k) \\ &= p_k^T \underbrace{(b - Ax_k)}_{=-r_k} \\ &= -p_k^T r_k \end{aligned}$$

Dividing the leftmost term and the rightmost term by $p_k^T A p_k$ we get the equality,

$$\frac{p_k^T A(x^* - x_0)}{p_k^T A p_k} = -\frac{p_k^T r_k}{p_k^T A p_k} \Rightarrow \sigma_k = \alpha_k. \tag{3.22}$$

In light of this relationship and (3.18), if we step along the conjugate directions using α_k, we will solve the linear system $Ax = b$ in at most n iterations.

There is a simple interpretation of the conjugate directions. If A is diagonal, the isosurfaces are ellipsoids with axes aligned with coordinate directions. We could then find minimum by performing univariate minimization along each coordinate direction in turn and this would result in convergence to the minimum in n iterations.

When A a positive-definite matrix that is not diagonal, its contours are still elliptical, but they are not aligned with the coordinate axes. Minimization along coordinate directions no longer leads to solution in n iterations (or even a finite n). Thus, we need a different set of search directions to recover convergence in n iterations: this is the role of the conjugate directions.

The original variables x are related to the new ones by $x = S\hat{x}$. Inverting this transformation,

$$\hat{x} = S^{-1}x, \tag{3.23}$$

where $S = \begin{bmatrix} p_0 & p_1 & \cdots & p_{n-1} \end{bmatrix}$, a matrix whose columns are the set of conjugate directions with respect to A. The quadratic now becomes

$$\hat{\phi}(\hat{x}) = \frac{1}{2}\hat{x}^T \left(S^T A S\right) \hat{x} - \left(S^T b\right)^T \hat{x} \tag{3.24}$$

By conjugacy, $\left(S^T A S\right)$ is diagonal so we can do a sequence of n line minimizations along the coordinate directions of $\hat{x}$. Each univariate minimization determines a component of x^* correctly.

When the conjugate-gradient method is adapted to general nonlinear problems, we obtain the *Fletcher–Reeves method.*

Algorithm 6 Nonlinear conjugate gradient method

Input: Initial guess, x_0, convergence tolerances, $\varepsilon_g, \varepsilon_a$ and ε_r.
Output: Optimum, x^*

```
k ← 0
repeat
  Compute the gradient of the objective function, g(x_k)
  if k=0 then
    Compute the normalized steepest descent direction, p_k ← −g(x_k)/‖g(x_k)‖
  else
    Compute β ← (g_k^T g_k) / (g_{k−1}^T g_{k−1})
    Compute the conjugate gradient direction p_k = −g_k/‖g(x_k)‖ + β_k p_{k−1}
  end if
  Perform line search to find step length α_k
  Update the current point, x_{k+1} ← x_k + α_k p_k
  k ← k + 1
until |f(x_k) − f(x_{k−1})| ≤ ε_a + ε_r |f(x_{k−1})| and ‖g(x_{k−1})‖ ≤ ε_g
```

Usually, a *restart* is performed every n iterations for computational stability, i.e. we start with a steepest descent direction.

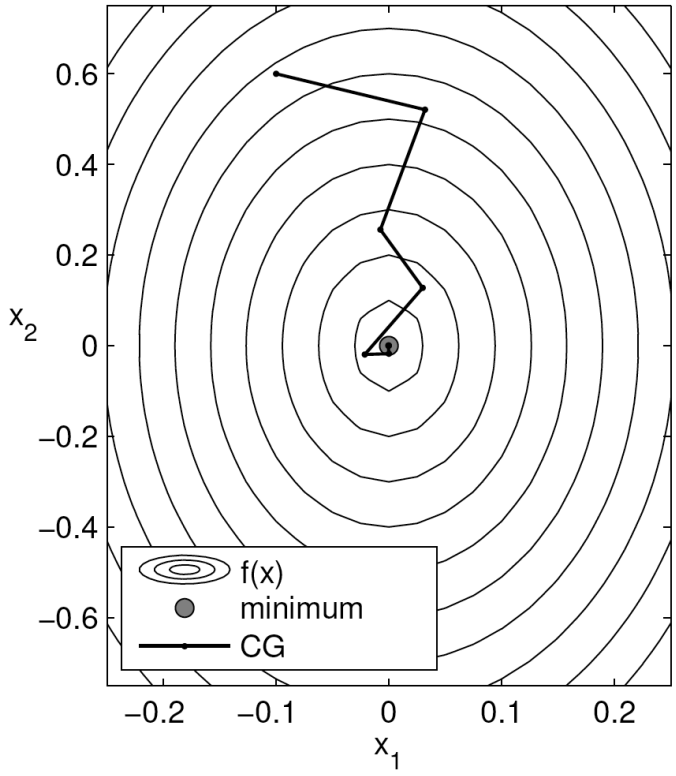


Figure 3.5: Solution path of the nonlinear conjugate gradient method.

The conjugate gradient method does not produce well-scaled search directions, so we can use same strategy to choose the initial step size as for steepest descent.

Several variants of the Fletcher–Reeves CG method have been proposed. Most of these variants differ in their definition of β_k. For example, Dai and Yuan [3] propose

$$\beta_k = \frac{\|g_k\|^2}{(g_k - g_{k-1})^T p_{k-1}}. \tag{3.25}$$

Another variant is the Polak–Ribière formula

$$\beta_k = \frac{g_k^T (g_k - g_{k-1})}{g_{k-1}^T g_{k-1}}. \tag{3.26}$$

The only difference relative to the steepest descent is that the each descent direction is modified by adding a contribution from the previous direction.

The convergence rate of the nonlinear conjugate gradient is linear, but can be superlinear, converging in n to $5n$ iterations.

Since this method is just a minor modification away from steepest descent and performs much better, there is no excuse for steepest descent!

Example 3.6. Conjugate Gradient Method Figure 3.5 plots the solution path of the nonlinear conjugate gradient method applied to the objective function f given by (3.11). The qualitative improvement over the steepest-descent path (Fig. 3.4) is evident.

3.6 Newton's Method

The steepest descent and conjugate gradient methods only use first order information (the first derivative term in the Taylor series) to obtain a local model of the function.

Newton methods use a second-order Taylor series expansion of the function about the current design point, i.e. a quadratic model

$$f(x_k + s_k) \approx f_k + g_k^T s_k + \frac{1}{2} s_k^T H_k s_k, \tag{3.27}$$

where s_k is the step to the minimum. Differentiating this with respect to s_k and setting it to zero, we can obtain the step for that minimizes this quadratic,

$$H_k s_k = -g_k. \tag{3.28}$$

This is a linear system which yields the *Newton step*, s_k, as a solution.

If f is a quadratic function and H_k is positive definite, Newton's method requires only one iteration to converge from any starting point. For a general nonlinear function, Newton's method converges quadratically if x_0 is sufficiently close to x^* and the Hessian is positive definite at x^*.

As in the single variable case, difficulties and even failure may occur when the quadratic model is a poor approximation of f far from the current point. If H_k is not positive definite, the quadratic model might not have a minimum or even a stationary point. For some nonlinear functions, the Newton step might be such that $f(x_k + s_k) > f(x_k)$ and the method is not guaranteed to converge.

Another disadvantage of Newton's method is the need to compute not only the gradient, but also the Hessian, which contains $n(n+1)/2$ second order derivatives.

A small modification to Newton's method is to perform a line search along the Newton direction, rather than accepting the step size that would minimize the quadratic model.

Algorithm 7 Modified Newton's method

Input: Initial guess, x_0, convergence tolerances, $\varepsilon_g, \varepsilon_a$ and ε_r.
Output: Optimum, x^*

```
k ← 0
repeat
  Compute the gradient of the objective function, g(x_k)
  Compute the Hessian of the objective function, H(x_k)
  Compute the search direction, p_k = −H^{−1} g_k
  Perform line search to find step length α_k, starting with α = 1
  Update the current point, x_{k+1} ← x_k + α_k p_k
  k ← k + 1
until |f(x_k) − f(x_{k−1})| ≤ ε_a + ε_r |f(x_{k−1})| and ‖g(x_{k−1})‖ ≤ ε_g
```

Although this modification increases the probability that $f(x_k + p_k) < f(x_k)$, it is still vulnerable to the problem of having an Hessian that is not positive definite and has all the other disadvantages of the pure Newton's method.

We could also introduce a modification to use a symmetric positive definite matrix instead of the real Hessian to ensure descent:

$$B_k = H_k + \gamma I,$$

where γ is chosen such that all the eigenvalues of B_k are greater than a tolerance $\delta > 0$ (if δ is too close to zero, B_k might be ill-conditioned, which can lead to round of errors in the solution of the linear system).

When using Newton's method, or quasi-Newton methods described next, the starting step length $\bar{\alpha}$ is usually set to 1, since Newton's method already provides a good guess for the step size.

The step size reduction ratio (ρ in the backtracking line search) sometimes varies during the optimization process and is such that $0 < \rho < 1$. In practice ρ is not set to be too close to 0 or 1.

Example 3.7. Modified Newton's Method In this example, we apply Algorithm 7 to the function f given by (3.11) starting from $(-0.1, 0.6)^T$. The resulting optimization path is shown in Fig. 3.6, where we can see the rapid convergence of the method: indeed, the first step is indistinguishable from subsequent steps, since f is almost a quadratic. However, moving the initial guess slightly away from the origin leads to divergence, because the Newton search direction ceases to be a descent direction.

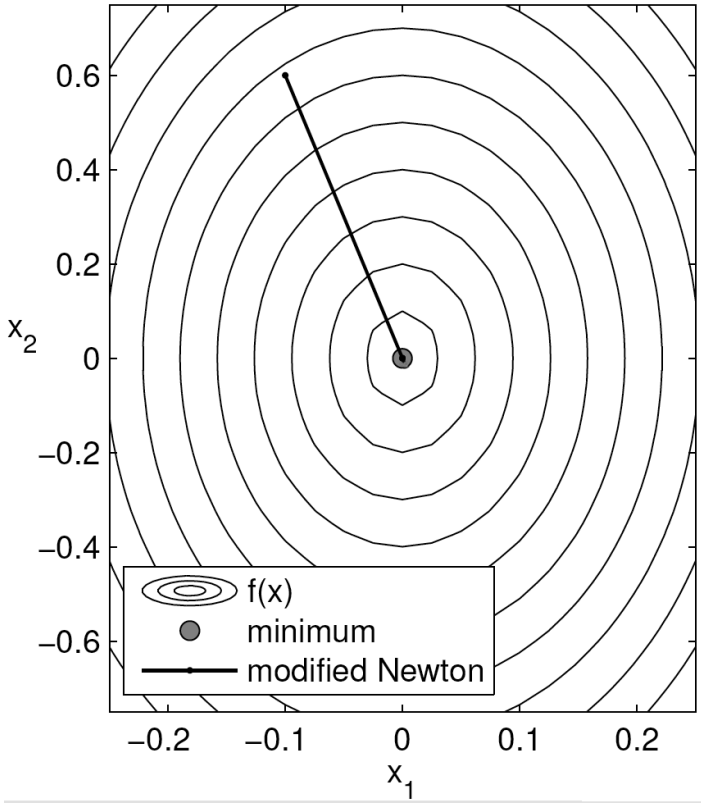


Figure 3.6: Solution path of the modified Newton's method.

3.7 Quasi-Newton Methods

This class of methods uses first order information only, but build second order information — an approximate Hessian — based on the sequence of function values and gradients from previous iterations. The various quasi-Newton methods differ in how they update the approximate Hessian. Most of these methods also force the Hessian to be symmetric and positive definite, which can greatly improve their convergence properties.

3.7.1 Davidon–Fletcher–Powell (DFP) Method

One of the first quasi-Newton methods was devised by Davidon in 1959, who a physicist at Argonne National Laboratories. He was using a coordinate descent method, and had limited computer resources, so he invented a more efficient method that resulted in the first quasi-Newton method.

This was one of the most revolutionary ideas in nonlinear optimization. Davidon's paper was not accepted for publication! It remained a technical report until 1991, when it was published as the first paper in the inaugural issue of the SIAM Journal on Optimization [4].

Fletcher and Powell later modified the method and showed that it was much faster than current ones [7], and hence it became known as the Davidon–Fletcher–Powell (DFP) method.

Suppose we model the objective function as a quadratic at the current iterate x_k:

$$\phi_k(p) = f_k + g_k^T p + \frac{1}{2} p^T B_k p, \tag{3.29}$$

where B_k is an $n \times n$ symmetric positive definite matrix that is *updated* every iteration. This is the same model we used previously, a quadratic expansion about our current iterate, except we now use an approximation of the Hessian B_k, rather than the true Hessian.

The step p_k that minimizes this convex quadratic model can be written as,

$$p_k = -B_k^{-1} g_k. \tag{3.30}$$

This solution is used to compute the search direction to obtain the new iterate

$$x_{k+1} = x_k + \alpha_k p_k \tag{3.31}$$

where α_k is obtained using a line search.

Instead of computing B_k "from scratch" at every iteration, a quasi-Newton method updates it in a way that accounts for the curvature measured during the most recent step. We want to build an updated quadratic model,

$$\phi_{k+1}(p) = f_{k+1} + g_{k+1}^T p + \frac{1}{2} p^T B_{k+1} p. \tag{3.32}$$

What requirements should we impose on B_{k+1}, based on the new information from the last step?

We would like our approximate Hessian to result in a quadratic model that matches both the function value f_{k+1} the gradient g_{k+1} at our new iterate x_{k+1}. Additionally, we would like the use the gradient information from our previous iterate and have our resulting quadratic model match the gradient at the location of the previous iterate g_k. In 1-dimension, the requirements of matching the slope and value at the current point, and the slope of the previous point are illustrated in Fig. 3.7.

The gradient of the quadratic model (3.32) is

$$\nabla \phi_{k+1}(p) = g_{k+1} + B_{k+1} p. \tag{3.33}$$

The first two requirements we get for free just by nature of building a quadratic expansion around x_{k+1}.

$$\phi_{k+1}(0) = f_{k+1} \tag{3.34}$$

$$\nabla \phi_{k+1}(0) = g_{k+1} \tag{3.35}$$

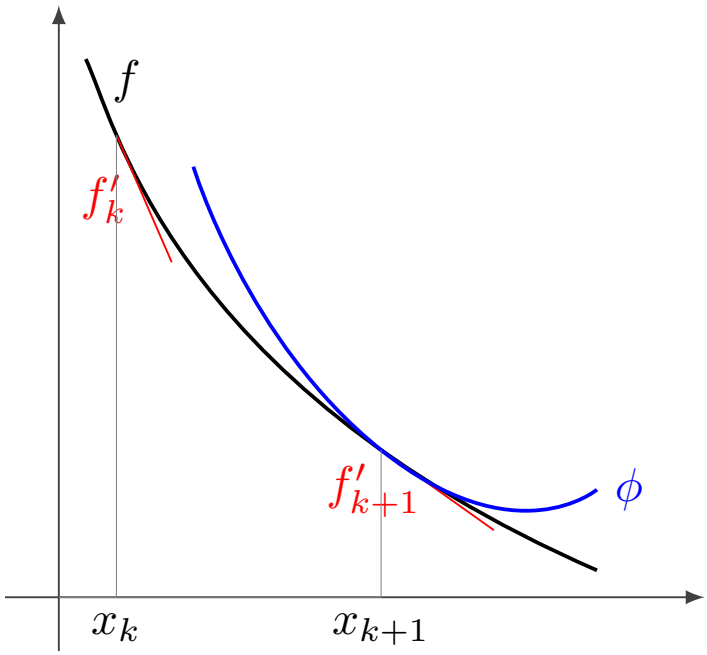


Figure 3.7: Projection of the quadratic model onto the last search direction, illustrating the secant condition

The later requirement that we match the gradient of the actual function at the previous point x_k implies,

$$\nabla\phi_{k+1}\left(-\alpha_k p_k\right) = g_{k+1} - \alpha_k B_{k+1} p_k = g_k. \tag{3.36}$$

Rearranging we obtain,

$$B_{k+1}\alpha_k p_k = g_{k+1} - g_k. \tag{3.37}$$

which is called the *secant condition*. Just as changes in function values can be used to estimate gradients, here we are using changes in gradients to estimate curvature.

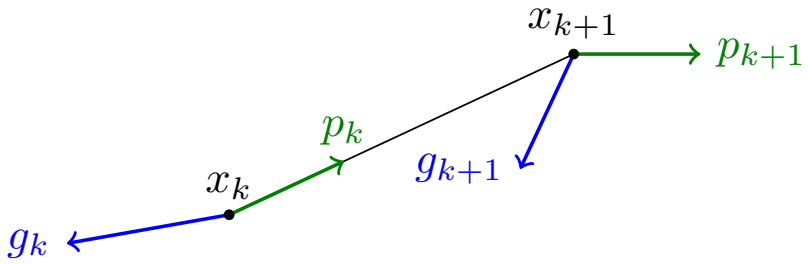


Figure 3.8: The difference in the gradients at two subsequent points g_k and g_{k+1} is projected onto the direction p_k to enforce the secant condition show in Fig. 3.7.

For convenience, we will set the difference of the gradients to $y_k = g_{k+1} - g_k$, and $s_k = \alpha_k p_k = x_{k+1} - x_k$ so the secant condition is then written as

$$B_{k+1} s_k = y_k. \tag{3.38}$$

We now have $n(n+1)/2$ unknowns (because the matrix is symmetric) and only $2n$ constraints (n from the secant condition, and n from requiring positive definiteness), so this system has an infinite number of solutions for B_{k+1}. To determine the solution uniquely, we impose a condition that among all the matrices that satisfy the secant condition, selects the B_{k+1} that is "closest" to the previous Hessian approximation B_k, i.e., we solve the following optimization problem,

$$\begin{aligned} \text{minimize} \quad & \|B - B_k\| \\ \text{with respect to} \quad & B \\ \text{subject to} \quad & B = B^T, \quad Bs_k = y_k. \end{aligned} \tag{3.39}$$

Using different matrix norms result in different quasi-Newton methods. One norm that makes it easy to solve this problem and possesses good numerical properties is the weighted Frobenius norm

$$\|A\|_W = \|W^{1/2} A W^{1/2}\|_F, \tag{3.40}$$

where the norm is defined as $\|C\|_F = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij}^2$. The weights W are chosen to satisfy certain favorable conditions. The norm is adimensional (i.e., does not depend on the units of the problem) if the weights are chosen appropriately. For further details on the derivation of the quasi-Newton methods, see Dennis and Moré's review [5].

Using this norm and weights, the unique solution of the norm minimization problem (3.39) is,

$$B_{k+1} = \left(I - \frac{y_k s_k^T}{y_k^T s_k}\right) B_k \left(I - \frac{s_k y_k^T}{y_k^T s_k}\right) + \frac{y_k y_k^T}{y_k^T s_k}, \tag{3.41}$$

which is the *DFP updating formula* originally proposed by Davidon.

Using the inverse of B_k is usually more useful, since the search direction can then be obtained by matrix multiplication. Defining,

$$V_k = B_k^{-1}. \tag{3.42}$$

The DFP update for the inverse of the Hessian approximation can be shown to be

$$V_{k+1} = V_k - \frac{V_k y_k y_k^T V_k}{y_k^T V_k y_k} + \frac{s_k s_k^T}{y_k^T s_k} \tag{3.43}$$

Note that this is a rank 2 update. A rank one update is $B_{k+1} = B_k + uv^T$, which adds a matrix whose columns are the same vector u multiplied by the scalars in v. this means that the columns are all linearly dependent and the matrix spans 1-space and is rank 1. A rank two update looks like $B_{k+1} = B_k + u_1 v_1^T + u_2 v_2^T$ and the update spans 2-space.

Algorithm 8 Quasi-Newton method with DFP update

Input: Initial guess, x_0, convergence tolerances, $\varepsilon_g, \varepsilon_a$ and ε_r.
Output: Optimum, x^*

$k \leftarrow 0$
$V_0 \leftarrow I$
repeat
  Compute the gradient of the objective function, $g(x_k)$
  Compute the search direction, $p_k \leftarrow -V_k g_k$
  Perform line search to find step length α_k, starting with $\alpha \leftarrow 1$
  Update the current point, $x_{k+1} \leftarrow x_k + \alpha_k p_k$
  Set the step length, $s_k \leftarrow \alpha_k p_k$
  Compute the change in the gradient, $y_k \leftarrow g_{k+1} - g_k$
  $A_k \leftarrow \frac{V_k y_k y_k^T V_k}{y_k^T V_k y_k}$
  $B_k \leftarrow \frac{s_k s_k^T}{s_k^T y_k}$
  Compute the updated approximation to the inverse of the Hessian, $V_{k+1} \leftarrow V_k - A_k + B_k$
until $|f(x_k) - f(x_{k-1})| \leq \varepsilon_a + \varepsilon_r |f(x_{k-1})|$ and $\|g(x_{k-1})\| \leq \varepsilon_g$

3.7.2 Broyden–Fletcher–Goldfarb–Shanno (BFGS) Method

The DFP update was soon superseded by the BFGS formula, which is generally considered to be the most effective quasi-Newton update. The key idea is that we don't actually care about finding B_k, but rather as suggested by Eq. (6.10) we really care about its inverse so that we can compute the search direction. Rather than estimate B_k and then solve a linear system, we can instead directly estimate its inverse V_k. The secant condition is then modified to

$$\begin{aligned} B_{k+1} s_k &= y_k \\ s_k &= B_{k+1}^{-1} y_k \\ s_k &= V_{k+1} y_k \end{aligned} \tag{3.44}$$

withe the resulting optimization problem.

$$\begin{aligned} \text{minimize} \quad & \|V - V_k\| \\ \text{with respect to} \quad & V \\ \text{subject to} \quad & V = V^T, \quad V y_k = S_k \end{aligned} \tag{3.45}$$

The solution is given by

$$V_{k+1} = \left[I - \frac{s_k y_k^T}{s_k^T y_k} \right] V_k \left[I - \frac{y_k s_k^T}{s_k^T y_k} \right] + \frac{s_k s_k^T}{s_k^T y_k}. \tag{3.46}$$

As noted above we can actually directly estimate V_{k+1} with the DFP method as well. The difference is that the DFP method is solving an optimization problem for B and then finding the corresponding inverse, where as the BFGS method is directly solving an optimization problem for V. The latter has shown to be generally more accurate and is the most popular quasi-Newton method.

We noted that the DFP or BFGS methods result in rank-2 updates to the previous Hessian estimate (or inverse Hessian estimate), but have not yet discussed how to choose the initial starting point. It is easily proved with either of these methods, that if we start with a positive definite

matrix then the updates will also remain positive definite. In the absence of other information, we generally start with the identity matrix (or a scaled version of the identity matrix) which corresponds to steepest descent on the first iteration.

Figure 3.9: Broyden, Fletcher, Goldfarb and Shanno at the NATO Optimization Meeting (Cambridge, UK, 1983), which was a seminal meeting for continuous optimization (courtesy of Prof. John Dennis)

Example 3.8. BFGS As for the previous methods, we apply BFGS to the function f given by (3.11). The solution path, starting at $(-0.1, 0.6)^T$, is shown in 3.10. Compare the path of BFGS with that from modified Newton's method (Fig. 3.6) and the conjugate gradient method (Fig. 3.5).

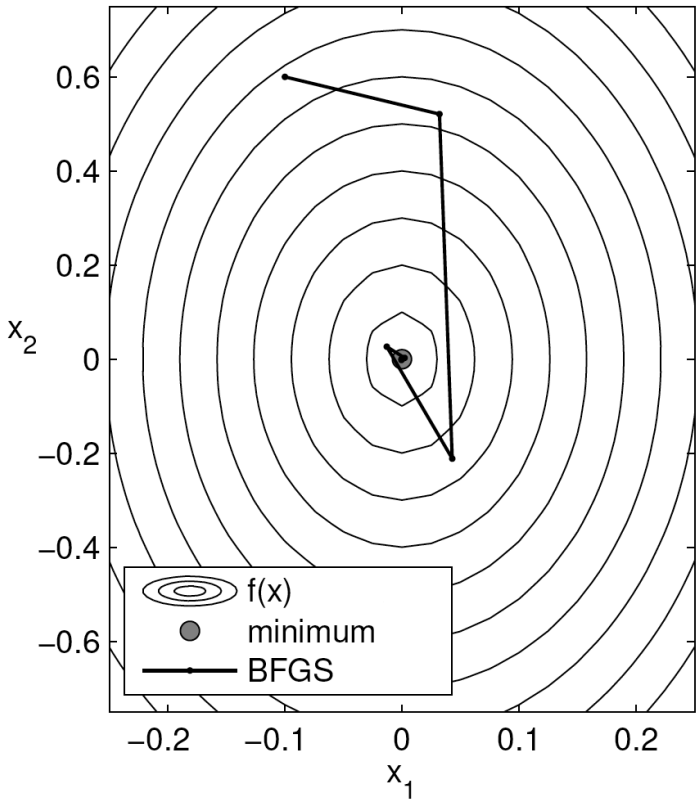


Figure 3.10: Solution path of the BFGS method.

Example 3.9. Minimization of the Rosenbrock Function

The following figures show the iteration path and convergence history of minimizing the Rosenbrock function

$$f(x) = 100\left(x_2 - x_1^2\right)^2 + (1 - x_1)^2\,, \tag{3.47}$$

starting from $x_0 = (-1.2, 1.0)^T$.

In Fig. 3.15 we see that Newton's method requires fewer iterations than a Quasi-Newton method as expected. However, we should also keep in mind that each iteration in Newton's method tends to be more expensive than a corresponding Quasi-Newton iterate. Newton's method requires computation of the full Hessian and requires a linear solve in Eq. (6.13), which is an $\mathcal{O}(n^3)$ operation. Quasi-Newton methods that directly use the inverse of an approximate Hessian need only compute a matrix-vector multiplication which is an $\mathcal{O}(n^2)$ operation.

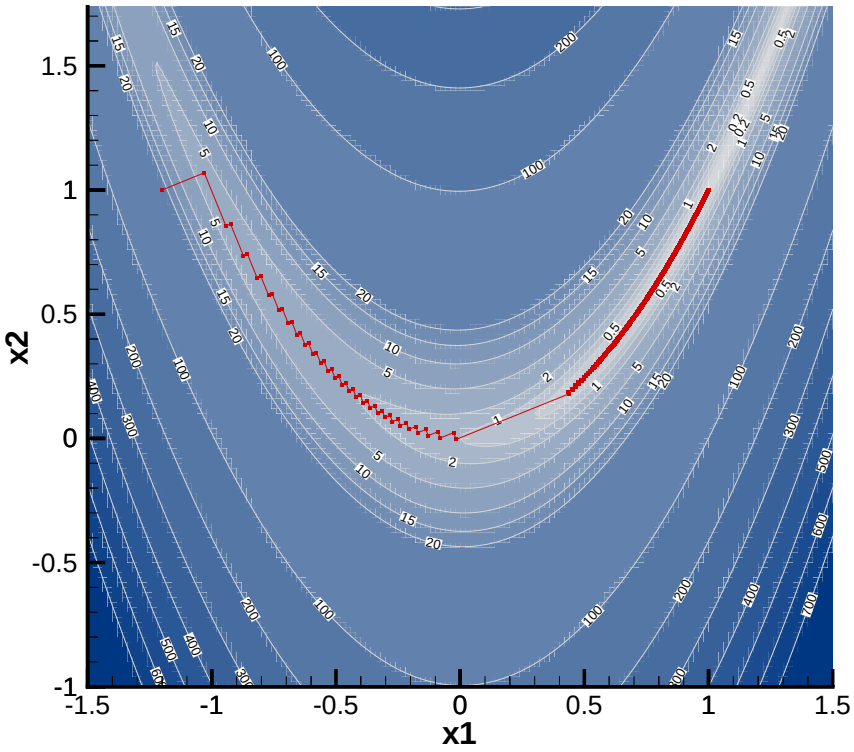


Figure 3.11: Solution path of the steepest descent method

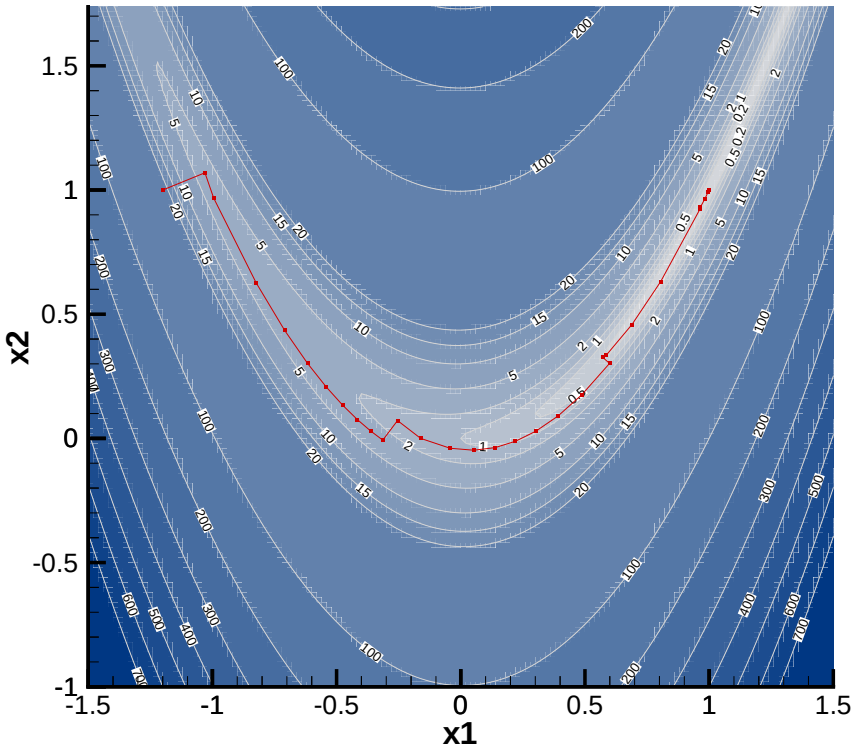


Figure 3.12: Solution path of the nonlinear conjugate gradient method

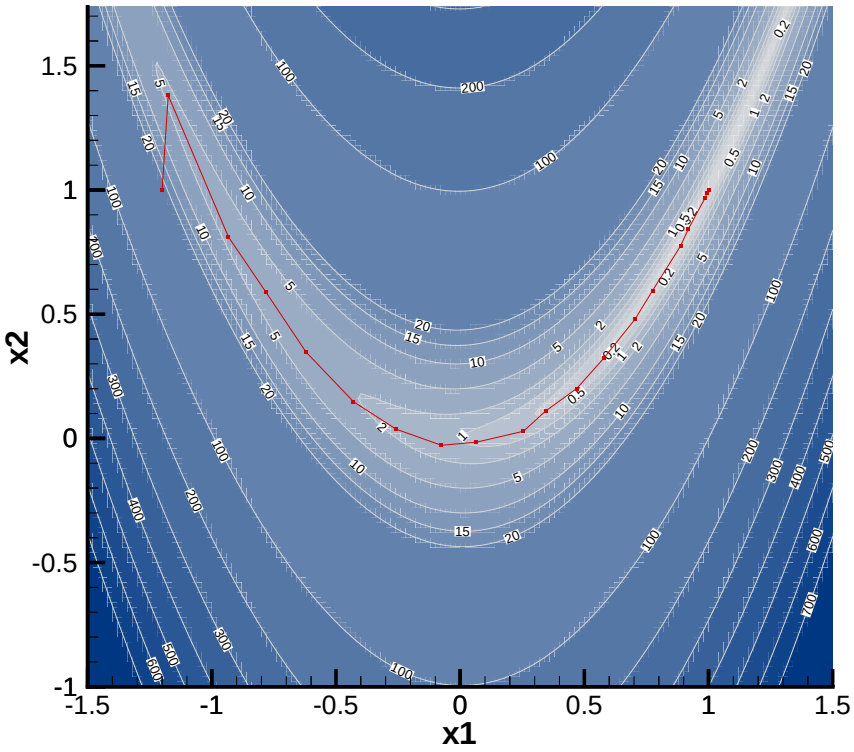


Figure 3.13: Solution path of the modified Newton method

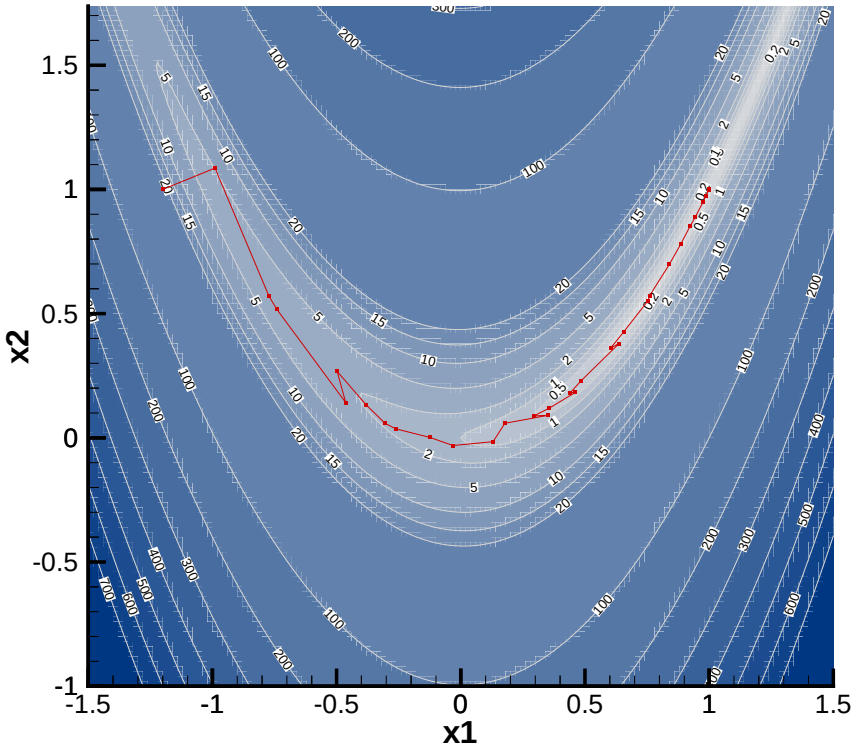


Figure 3.14: Solution path of the BFGS method

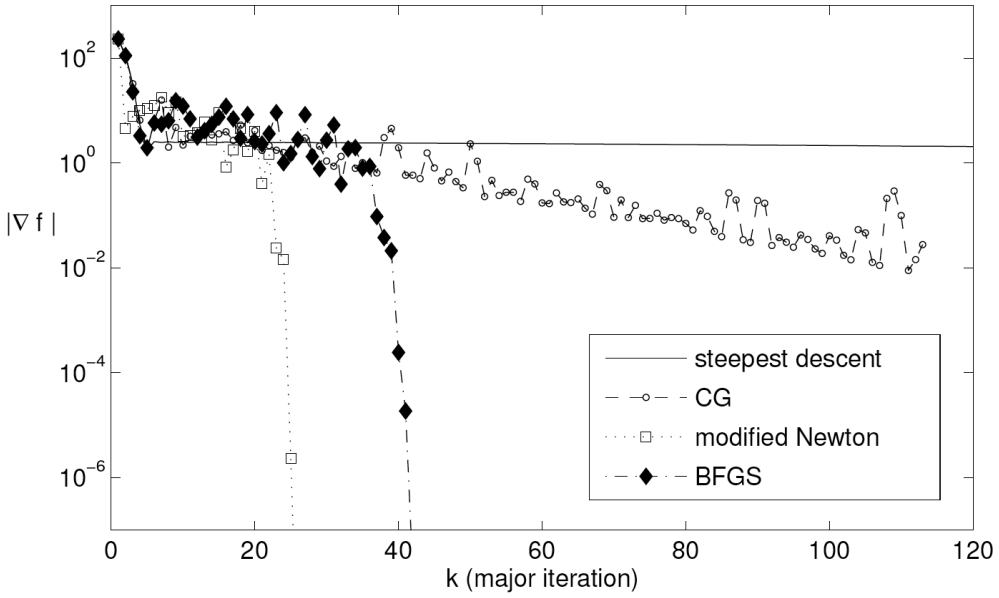


Figure 3.15: Comparison of convergence rates for the Rosenbrock function

3.7.3 Symmetric Rank-1 Update Method (SR1)

If we drop the requirement that the approximate Hessian (or its inverse) be positive definite, we can derive a simple rank-1 update formula for B_k that maintains the symmetry of the matrix and satisfies the secant equation. Such a formula is given by the symmetric rank-1 update (SR1) method (we use the Hessian update here, and not the inverse V_k):

$$B_{k+1} = B_k + \frac{(y_k - B_k s_k)(y_k - B_k s_k)^T}{(y_k - B_k s_k)^T s_k}. \tag{3.48}$$

With this formula, we must consider the cases when the denominator vanishes, and add the necessary safe-guards:

1. if $y_k = B_k s_k$ then the denominator is zero, and the only update that satisfies the secant equation is $B_{k+1} = B_k$ (i.e., do not change the matrix).

2. if $y_k \neq B_k s_k$ and $(y_k - B_k s_k)^T s_k = 0$ then there is no symmetric rank-1 update that satisfies the secant equation.

To avoid the second case, we update the matrix only if the following condition is met:

$$|y_k^T (s_k - B_k y_k)| \geq r \|s_k\| \|y_k - B_k s_k\|,$$

where $r \in (0, 1)$ is a small number (e.g., $r = 10^{-8}$). Hence, if this condition is not met, we use $B_{k+1} = B_k$.

Why would we be interested in a Hessian approximation that is potentially indefinite? In practice, the matrices produced by SR1 have been found to approximate the true Hessian matrix well (often better than BFGS). This may be useful in trust-region methods (see next section) or constrained optimization problems; in the latter case the Hessian of the Lagrangian is often indefinite, even at the minimizer.

To use the SR1 method in practice, it may be necessary to add a diagonal matrix γI to B_k when calculating the search direction, as was done in modified Newton's method. In addition, a simple back-tracking line search can be used, since the Wolfe conditions are not required as part of the update (unlike BFGS).

3.8 Trust Region Methods

Trust region, or "restricted-step" methods are a different approach to resolving the weaknesses of the pure form of Newton's method, arising from an Hessian that is not positive definite or from a function that is highly nonlinear.

In the line search approaches we have been discussing we first select a search direction then focus our efforts on performing an (approximate) line search along that direction. Trust region methods could be thought of as a reversal of those roles. We first decide on a maximum step size (the size of our trust region) then focus on performing an (approximate) minimization within that region. This minimization ultimately allows us to choose the direction and step size simultaneously.

In equation form the trust region problem is

$$\begin{aligned} \text{minimize} \quad & m(s) \\ \text{with respect to} \quad & s \\ \text{subject to} \quad & \|s\| \leq \Delta \end{aligned} \tag{3.49}$$

where $m(s)$ is the local trust region model. Generally, the model will be a quadratic function, but other models may also be considered. The norm is often the Euclidean norm, but other norms are sometimes used and in some cases can have a significant impact on convergence behavior. Note that we use the notation s instead of p to indicate that this is a step vector (direction and magnitude) and not just a step direction p.

With a quadratic trust-region model and the Euclidean norm we can redefine our trust-region subproblem as

$$\begin{aligned} \text{minimize} \quad & m(s) = f_k + g_k^T s + \frac{1}{2} s^T B_k s \\ \text{subject to} \quad & \|s\|_2 \leq \Delta_k \end{aligned} \tag{3.50}$$

where B_k is the (approximate) Hessian at our current iterate. This is a quadratically constrained quadratic program (QCQP). If the problem was unconstrained and B was positive definite we could jump to the solution $s = -B_k^{-1} g_k$, like we do in the line search methods. For the QCQP, however, we do not have an analytic solution. While the problem is still straightforward to solve numerically, it requires an iterative solution approach with multiple factorizations. For the same reasons why we only perform an approximate line search in the line search approach, in the trust region approach we also usually resort to an *approximate* solution to the QCQP. Additionally, the inclusion of the trust-region constraint allows us to can relax the requirement that B_k is positive definite.

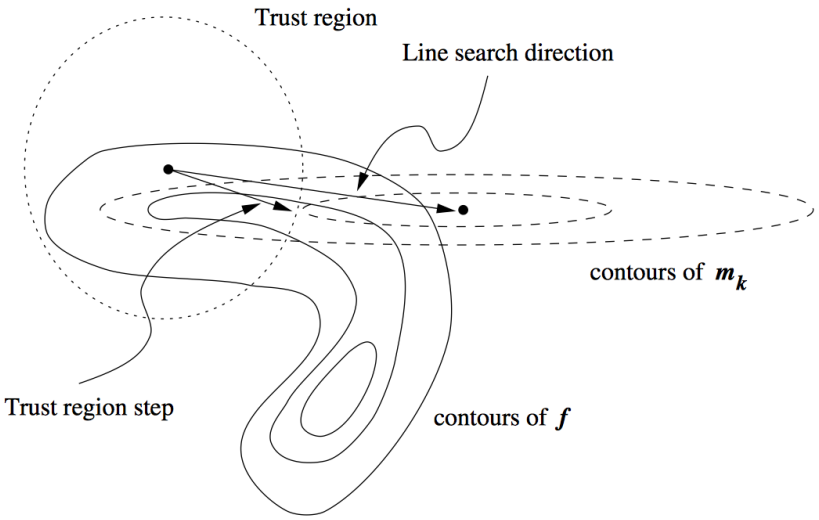


Figure 3.16: An example function (contours), with a local quadratic model, and a spherical trust region. The two arrows show the minimum of the quadratic which would be our search-direction in a Newton or Quasi-Newton line search method, and the solution for our constrained optimization problem that seeks for the minimum of the quadratic function within the trust region radius.

Figure 3.16 shows an example of function contours, a local quadratic model of the function, and a spherical trust-region. From the figure we note that as the trust region size changes the solution to Eq. (3.50) also changes. Unlike line search methods, as the trust region size shrinks the direction of our step will in general change.

3.8.1 Trust Region Sizing

Before discussing strategies for approximately solving Eq. (3.50) we will present an overview of a trust-region algorithm that describes how to shrink and grow the trust region. The metric we use to assess the appropriateness of the trust region size is the reliability index, r_k:

$$r_k = \frac{f(x_k) - f(x_k + s_k)}{m(0) - m(s_k)}. \tag{3.51}$$

The denominator is the predicted reduction which will always be nonnegative. The numerator is the actual reduction. A r_k value close to 1 means that the trust region model agrees very well with the actual function. A r_k value larger than 1 is fortuitous and means the actual decrease was even greater than expected. Negative values of r_k are clearly bad as they mean that the function actually increased at our expected minimum. A general procedure is outlined below

Algorithm 9 Trust region algorithm

Input: Initial guess x_0, convergence tolerances, $\varepsilon_g, \varepsilon_a$ and ε_r, initial size of the trust region, Δ_0
Output: Optimum, x^*

```
k ← 0
repeat
  Compute the Hessian of the objective function H(x_k), and (approximately) solve the quadratic
  subproblem (Eq. (3.50))
  Evaluate the reliability index r_k (Eq. (3.51))
  if r_k < 0.25 then
    Δ_{k+1} ← Δ_k/4 {Model is poor; shrink the trust region}
  else if r_k > 0.75 and ||s_k|| = Δ_k then
    Δ_{k+1} ← min(2Δ_k, Δ_max) {Model is good and new point on edge; expand trust region}
  else
    Δ_{k+1} ← Δ_k {New point is within trust region and/or the model is reasonable; keep trust
    region the same size}
  end if
  if r_k ≤ 0 then
    x_{k+1} ← x_k {Reject step; keep trust region centered about the same point}
  else
    x_{k+1} ← x_k + s_k {Move center of trust region to new point}
  end if
  k ← k + 1
until |f(x_k) − f(x_{k−1})| ≤ ε_a + ε_r|f(x_{k−1})| and/or ||g(x_{k−1})|| ≤ ε_g
```

The initial value of Δ is usually 1. We may choose to implement a Δ_{max} that we will not allow the trust region to expand beyond. The same stopping criteria used in other gradient-based methods are applicable.

One way to interpret these problems is to say that they arise from the fact that we are stepping outside a the region for which the quadratic approximation is reasonable. Thus we can overcome this difficulties by minimizing the quadratic function within a region around x_k within which we *trust* the quadratic model. When our model is performing well, we expand the trust region. When it performs poorly we shrink the trust region. If we shrink the trust region sufficiently, our local model should compare well with the underlying function.

3.8.2 Solution of the Trust-Region Subproblem

In line search methods we sought approximate solutions to the line search in order to reduce computational expense, and even with very loose criteria the approximate solutions methods were provably convergent. For similar reasons we seek an approximate solution to the trust-region subproblem (Eq. (3.50)). These approximate solution methods also have corresponding convergence proofs, but they will not be detailed here. Instead we present an overview of some strategies used to approximately solve the quadratic trust-region subproblem.

Cauchy Point

The simplest approximate solution approach is to use a linearized version of Eq. (3.50) and the solution is known as the Cauchy point. The linearized problem is

$$\begin{aligned} \text{minimize} \quad & m(s) = f_k + g_k^T s \\ \text{subject to} \quad & \|s\|_2 \leq \Delta_k \end{aligned} \tag{3.52}$$

We can write out an analytic solution for this problem (where the k subscript is dropped for simplicity):

$$\begin{aligned} s &= \min\left(-\frac{g^T g}{g^T B g} g,\ -\Delta \frac{g}{\|g\|}\right), \quad \text{if } g^T B g > 0 \\ &= -\Delta \frac{g}{\|g\|}, \quad \text{otherwise} \end{aligned} \tag{3.53}$$

In all cases we see that the minimum solution lies along the $-g$ direction. If B is not positive definite then the minimum must lie at the boundary (locally the function looks like an upside-down bowl). The solution is then to move as far as possible in the negative gradient direction until we hit the trust region bound. For the case where B is positive definite then there are two possibilities: the minimum lies in the interior which corresponds to $-\frac{g^T g}{g^T B g} g$, or the minimum is at the boundary.

It should not be surprising that the method is not efficient. We have thrown out the curvature information and the resulting method is equivalent to the steepest descent method but with a step length dictated by the trust region sizing. The importance of the method is that it does have an associated convergence proof, and any methods that use at least some fixed multiple of the decrease of the Cauchy step are also provably convergent (similar to the conditions on provable convergence in line search methods).

Dogleg Method

A small improvement to the Cauchy point method is to find the minimum along a line segment from the current iterate to the Cauchy point and from the Cauchy point to the full unconstrained step (Fig. 3.17). The intuition is that if the trust region is very large then the solution would correspond to the unconstrained step $s = -B^{-1}g$, and if the trust-region was very small then the quadratic term would be negligible and the solution would approximately be $s \approx -\Delta \frac{g}{\|g\|}$. As the trust region size shrinks, the actual solution follows a curved path. he dogleg approach attempts to approximate this curved path through two line segments where the first ends at the minimizer in the steepest descent direction.

We can prove that the dogleg path intersects the trust-region boundary at one point at most. This allows for a easy solution algorithm as seen in Algorithm 10. In practice this heuristic approach is particularly effective, except for with convex functions (B is always positive semidefinite).

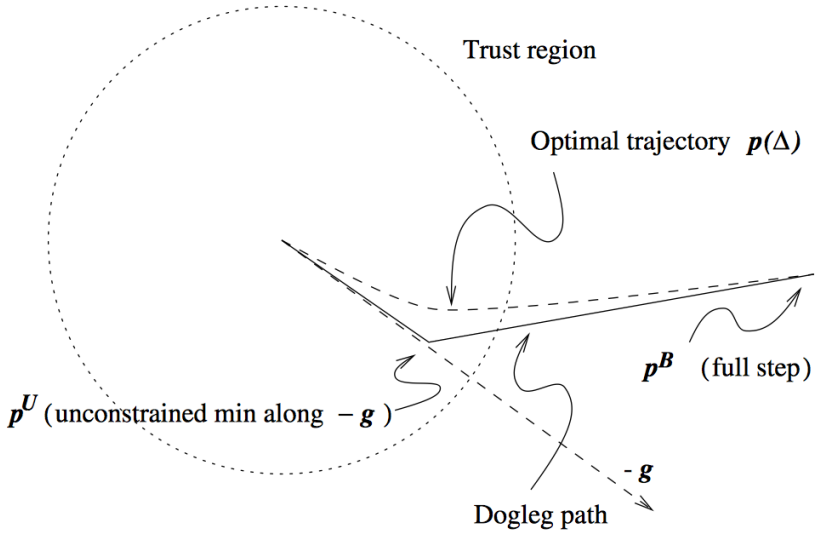


Figure 3.17: s

Algorithm 10 Dogleg update method

Input: Information at current iterate: x_k, g_k, B_k
Output: Location of next iterate (approximate solution): x_{k+1}

```
s_B = −B^{-1} g
if ‖s_B‖ < Δ then
    s = s_B
else
    s_U = −(g^T g)/(g^T B g) g
    Solve quadratic equation for τ: ‖s_U + (τ − 1)(s_B − s_U)‖^2 = Δ^2
    if 0 ≤ τ ≤ 1 then
        s = τ s_U
    else if 1 < τ ≤ 2 then
        s = s_U + (τ − 1)(s_B − s_U)
    end if
end if
x_{k+1} ← x_k + s
```

Two-dimensional Subspace Minimization

A more effective improvement to these two methods is to open up our search space not just to a line, but to the two-dimensional subspace spanned by the two vectors of interest: p_U and p_B or equivalently by g and $B^{-1}g$. The resulting optimization problem is

$$\begin{aligned} \text{minimize} \quad & m(s) = f_k + g_k^T s + \frac{1}{2} s^T B_k s \\ \text{subject to} \quad & \|s\|_2 \leq \Delta_k \\ & s \in \text{span}[g_k, B_k^{-1} g_k] \end{aligned} \tag{3.54}$$

This optimization problem results in a fourth degree polynomial that is inexpensive to solve. For further details the interested reader is referred to [2] and [16]. The inexact solution using this approach is often very close to the exact solution and requires only one matrix factorization.

Iterative Solution to Subproblem

A fourth technique applies Newton's method to approximately solve the trust-region subproblem. Instead of applying Newton's method until convergence, the number of iterations is limited to around three. The previous methods, by contrast require only one matrix factorization. This approach trades-off computational expensive for a better solution to the subproblem. This is generally only a worthwhile trade if the dimensionality of the problem is not very large and if the user supplies an exact Hessian. Because of these limitations we will not elaborate further on the details of these approaches, but numerous references exist [6, 9, 12, 17].

3.8.3 Comparison with Line Search Methods

Generally speaking, trust-region methods are more strongly dependent on accurate Hessians as compared to line search methods. For this reason, they are usually only effective if exact gradients can be supplied. Trust-region methods generally require fewer iterations than Quasi-Newton methods but each iteration is more computationally expensive because of the need for at least one matrix factorization.

Scaling can also be more challenging with trust-region approaches. Newton's method is invariant with scaling, but the use of a Euclidean trust-region constraint implicitly assumes that the function changes in each direction at a similar rate. Some enhancements try to address this issue through the use of elliptical trust regions rather than spherical ones.

Bibliography

[1] Ashok D. Belegundu and Tirupathi R. Chandrupatla. *Optimization Concepts and Applications in Engineering*, chapter 3. Prentice Hall, 1999.

[2] Richard H. Byrd, Robert B. Schnabel, and Gerald A. Shultz. Approximate solution of the trust region problem by minimization over two-dimensional subspaces. *Mathematical Programming*, 40-40(1-3):247–263, jan 1988. doi:10.1007/bf01580735. URL http://dx.doi.org/10.1007/bf01580735.

[3] Y.H. Dai and Y. Yuan. A nonlinear conjugate gradient with a strong global convergence property. *SIAM Journal on Optimization*, 10(1):177–182, 1999.

[4] William C. Davidon. Variable metric method for minimization. *SIAM Journal on Optimization*, 1(1):1–17, February 1991.

[5] J.E. Dennis and J. J. Moreé. Quasi-Newton methods, motivation and theory. *SIAM Review*, 19(1):46–89, 1977.

[6] Jennifer B. Erway, Philip E. Gill, and Joshua D. Griffin. Iterative methods for finding a trust-region step. *SIAM Journal on Optimization*, 20(2):1110–1131, jan 2009. doi:10.1137/070708494. URL http://dx.doi.org/10.1137/070708494.

[7] R. Fletcher and M. J. D. Powell. A rapidly convergent descent method for minimization. *The Computer Journal*, 6(2):163–168, 1963. doi:10.1093/comjnl/6.2.163.

[8] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization*, chapter 4. Academic Press, 1981.

[9] Jorge J. Moré and D. C. Sorensen. Computing a trust region step. *SIAM J. Sci. and Stat. Comput.*, 4(3):553–572, sep 1983. doi:10.1137/0904038. URL http://dx.doi.org/10.1137/0904038.

[10] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*, chapter 3. Springer-Verlag, 1999.

[11] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*, chapter 8. Springer-Verlag, 1999.

[12] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*, chapter 4. Springer, 2nd edition, 2006.

[13] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*, chapter 5. Springer, 2nd edition, 2006.

[14] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*, chapter 6. Springer, 2nd edition, 2006.

[15] Chinyere Onwubiko. *Introduction to Engineering Design Optimization*, chapter 4. Prentice Hall, 2000.

[16] Gerald A. Shultz, Robert B. Schnabel, and Richard H. Byrd. A family of trust-region-based algorithms for unconstrained minimization with strong global convergence properties. *SIAM J. Numer. Anal.*, 22(1):47–67, feb 1985. doi:10.1137/0722003. URL http://dx.doi.org/10.1137/0722003.

[17] Trond Steihaug. The conjugate gradient method and trust regions in large scale optimization. *SIAM J. Numer. Anal.*, 20(3):626–637, jun 1983. doi:10.1137/0720042. URL http://dx.doi.org/10.1137/0720042.

Chapter 4

Computing Derivatives

4.1 Introduction

The computation of derivatives is part of the broader field of sensitivity analysis. Sensitivity analysis is the study of how the outputs of a model change in response to changes in its inputs, and plays a key role in gradient-based optimization, uncertainty quantification, error analysis, model development, and computational model-assisted decision making. There are various types of sensitivities that can be defined. One common classification distinguishes between local and global sensitivity analysis [60]. Global sensitivity analysis aims to quantify the response with respect to inputs over a wide range of values, and it is better suited for models that have large uncertainties. Local sensitivity analysis aims to quantify the response for a fixed set of inputs, and is typically used in physics based models where the uncertainties tend to be lower. In this chapter, we focus on the computation of local sensitivities in the form of *first-order total derivatives*, where the model is a numerical algorithm representing a deterministic model. Although stochastic models require approaches that are beyond the scope of this chapter, some of these approaches can benefit from the deterministic techniques presented herein. While we focus on first-order derivatives, the techniques that we present can all be extended to compute higher-order derivatives. This incurs an additional computational cost that is prohibitive for large numbers of either inputs or outputs.

In the engineering optimization literature, the term "sensitivity analysis" is often used to refer to the computation of derivatives. While this is not incorrect, we prefer to use the more restrictive term. In addition, derivatives are sometimes referred to as "sensitivity derivatives" [5, 61, 69] or "design sensitivities" [4, 25, 67].

Derivatives play a central role in several numerical algorithms. In our context, we are particularly interested in their application to compute the gradients for the gradient-based optimization techniques presented in Chapter 3. The computation of derivatives has many other uses, such as Newton–Krylov methods applied to the solution of the Euler equations [26], coupled aerostructural equations [9, 10, 31], and quasi-Newton methods used to solve optimization problems [14, 19]. Other applications of derivatives include gradient-enhanced surrogate models [13], structural topology optimization [28, 33, 34, 63], aerostructural optimization [32, 47] and aircraft stability [39, 40].

The accuracy of the derivative computation affects the convergence behavior of the solver used in the algorithm. In the case of gradient-based optimization, accurate derivatives are important to ensure robust and efficient convergence, especially for problems with large numbers of constraints. The precision of the gradients limits that of the optimum solution, and inaccurate gradients can cause the optimizer to halt or to take a less direct route to the optimum that involves more iterations.

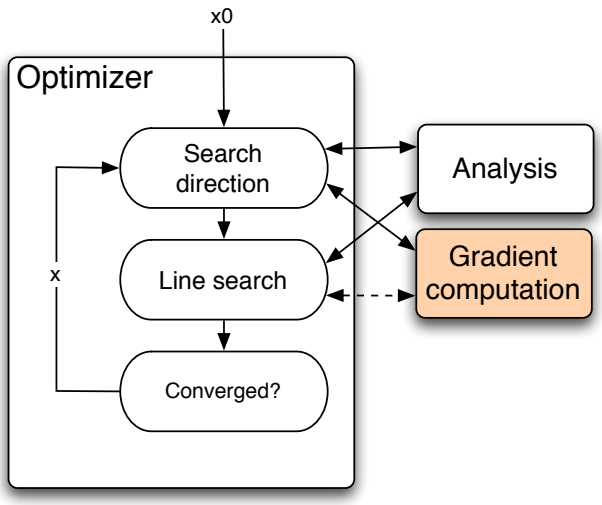


Figure 4.1: Dependence of gradient-based algorithms on the derivative computation

In this chapter, we assume that the numerical models are algorithms that solve a set of governing equations to find the state of a system. The computational effort involved in these numerical models, or simulations, is assumed to be significant. Examples of such simulations include computational fluid dynamics (CFD) and structural finite-element solvers.

Consider a general constrained optimization problem of the form:

$$\begin{aligned} \text{minimize} \quad & f(x_i) && && (4.1)\\ \text{w.r.t} \quad & x_i && i = 1, 2, \ldots, n && (4.2)\\ \text{subject to} \quad & c_j(x_i) \geq 0, && j = 1, 2, \ldots, m && (4.3) \end{aligned}$$

In order to solve this problem using a gradient-based optimization algorithm we usually require:

- The gradient of the objective function, $\nabla f(x) = \partial f / \partial x_i$, which is an n-vector.
- The gradient of all the active constraints at the current design point $\partial c_j / \partial x_i$, which is an $(m \times n)$ matrix.

Most gradient-based optimizers use finite-differences by default to compute the gradients. This is both costly and subject to inaccuracies. When the cost of calculating the gradients is proportional to the number of design variables, and this number is large, the computation of the derivatives usually becomes the bottleneck in the optimization cycle. In addition, inaccuracies in the derivatives due to finite differences are often the culprit in cases where gradient-based optimizers fail to converge.

4.1.1 Methods for Computing Derivatives

Finite Differences: very popular; easy, but lacks robustness and accuracy; run solver n times.

Complex-Step Method: relatively new; accurate and robust; easy to implement and maintain; run solver n times.

Symbolic Differentiation: accurate; restricted to explicit functions of low dimensionality.

Algorithmic/Automatic/Computational Differentiation: accurate; ease of implementation and cost varies.

(Semi)-Analytic Methods: efficient and accurate; long development time; *cost can be independent of n.*

4.2 Symbolic Differentiation

Symbolic differentiation is only possible for explicit functions, and can either be done by hand or by appropriate software, such as Maple, Mathematica and Matlab. For a sequence of composite functions, it is possible to use the chain rule, but for general algorithms it may be impossible to produce a closed form expression for the derivatives.

Nevertheless, symbolic differentiation is useful in the context of computational models, since it often can be applied to simple components of the model, reducing the computational cost of obtaining certain partial derivatives that are needed in other methods.

4.3 Finite Differences

Finite-difference methods are widely used to compute derivatives due to their simplicity and the fact that they can be implemented even when a given component is a black box. As previously mentioned, most gradient-based optimization algorithms perform finite-differences by default when the user does not provide the required gradients. Although these finite differences are neither particularly accurate or efficient, their biggest advantage is that they are extremely easy to implement.

Finite-difference formulas are derived from combining Taylor series expansions. Using the right combinations of these expansions, it is possible to obtain finite-difference formulas that estimate an arbitrary order derivative with any required order truncation error. The simplest finite-difference formula can be directly derived from a Taylor series expansion,

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + \ldots, \tag{4.4}$$

Solving for f' we get the finite-difference formula,

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h), \tag{4.5}$$

where h is the *finite-difference interval*. This approximation is called a *forward difference* and is directly related to the definition of derivative. The truncation error is $\mathcal{O}(h)$, and hence this is a first-order approximation.

In general there are multiple functions of interest, and thus $\boldsymbol{f}$ can be a vector that includes all of the outputs of a given component. The application of this formula requires the evaluation of a component at the reference point $\boldsymbol{x}$, and one perturbed point $\boldsymbol{x} + \boldsymbol{e}_j h$, and yields one column of the Jacobian (4.42). Each additional column requires an additional evaluation of the component. Hence, the cost of computing the complete Jacobian is proportional to the number of input variables of interest, n_x. The n-dimensional version of a forward step finite difference is:

$$\frac{\partial \boldsymbol{F}}{\partial x_j} = \frac{\boldsymbol{F}(\boldsymbol{x} + \boldsymbol{e}_j h) - \boldsymbol{F}(\boldsymbol{x})}{h} + \mathcal{O}(h) \tag{4.6}$$

$f(x+h)$	$+1.2345678901234\color{red}{31}$
$f(x)$	$+1.2345678901234\color{red}{56}$
Δ f	-0.000000000000025

Table 4.1: Subtractive cancellation

For a second-order estimate we can use the expansion of $f(x-h)$,

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2!}f''(x) - \frac{h^3}{3!}f'''(x) + \ldots, \tag{4.7}$$

and subtract it from the expansion (4.4). The resulting equation can then be solved for the derivative of f to obtain the *central-difference* formula,

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2). \tag{4.8}$$

More accurate estimates can also be derived by combining different Taylor series expansions.

Formulas for estimating higher-order derivatives can be obtained by nesting finite-difference formulas. We can use, for example the central difference (4.8) to estimate the second derivative instead of the first,

$$f''(x) = \frac{f'(x+h) - f'(x-h)}{2h} + \mathcal{O}(h^2). \tag{4.9}$$

and use central difference again to estimate both $f'(x+h)$ and $f'(x-h)$ in the above equation to obtain,

$$f''(x) = \frac{f(x+2h) - 2f(x) + f(x-2h)}{4h^2} + \mathcal{O}(h). \tag{4.10}$$

When estimating derivatives using finite-difference formulas we are faced with the *step-size dilemma*, that is, the desire to choose a small step size to minimize truncation error while avoiding the use of a step so small that errors due to subtractive cancellation become dominant.

Subtractive cancellation is due to finite precision arithmetic. In Table 4.1, for example, we show a case where we have 16-digit precision, and the step size was made small enough that the reference value and the perturbed value differ only in the last two digits. This means that when we do the subtraction required by the finite-difference formula, the final number only has a 2-digit precision. If h is made small enough, the difference vanishes to zero.

For functions of several variables, that is when x is a vector, then we have to calculate each component of the gradient $\nabla f(x)$ by perturbing the corresponding variable x_i.

The cost of calculating a gradient with finite-differences is therefore proportional to the number of design variables and f must be calculated for each perturbation of x_i. This means that if we use forward differences, for example, the cost would be $n+1$ times the cost of calculating f.

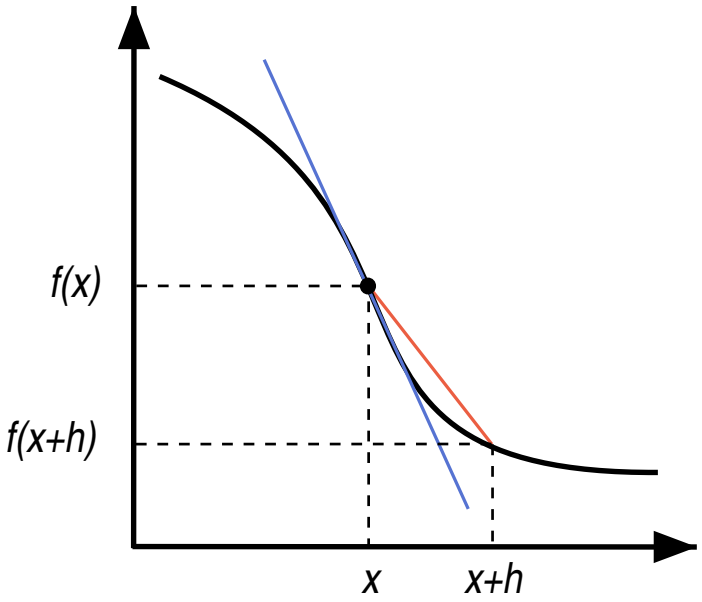


Figure 4.2: Graphical representation of the finite difference approximation

4.4 The Complex-Step Derivative Approximation

4.4.1 Origins

The complex-step derivative approximation, strangely enough, computes derivatives of real functions using complex variables. This method originated with the work of Lyness and Moler [37] and Lyness [38]. They developed several methods that made use of complex variables, including a reliable method for calculating the n^{th} derivative of an analytic function. However, only later was this theory rediscovered by Squire and Trapp [64], who derived a simple formula for estimating the first derivative. This estimate is suitable for use in modern numerical computing and has shown to be very accurate, extremely robust and surprisingly easy to implement, while retaining a reasonable computational cost [43, 46].

The first application of this approach to an iterative solver is due to Anderson et al. [3], who used it to compute derivatives of a Navier–Stokes solver, and later multidisciplinary systems [51]. Martins et al. [46] showed that the complex-step method is generally applicable to any algorithm and described the detailed procedure for its implementation. They also present an alternative way of deriving and understanding the complex step, and connect this to algorithmic differentiation.

The complex-step method requires access to the source code of the given component, and thus cannot be applied to black box components without additional effort in most cases. To implement the complex-step method, the source code of the component must be modified so that all real variables and computations are replaced with complex ones. In addition, some intrinsic functions need to be replaced, depending on the programming language. Martins et al. [46] provide a script that facilitates the implementation of the complex-step method to Fortran codes, as well as details for implementation in Matlab, C/C++ and Python.

The complex-step approach is now widely used, with applications ranging from the verification of high-fidelity aerostructural derivatives [32, 48] to development of immunology models [36]. In one case, the complex-step was implemented in an aircraft design optimization framework that involves multiple languages (Python, C, C++, and Fortran) [66], demonstrating the flexibility of

this approach. Other diverse applications include Jacobian matrices in liquid chromatography [**?**], first and second derivatives in Kalman filters [**?**], Hessian matrices in statistics [**?**], local sensitivities in biotechnology [**?**], and a Newton solver for turbulent flows [**?**].

4.4.2 Theory

We will now see that a very simple formula for the first derivative of real functions can be obtained using complex calculus. The complex-step derivative approximation, like finite-difference formulas, can also be derived using a Taylor series expansion. Rather than using a real step h, we now use a pure imaginary step, ih. If f is a real function in real variables and it is also analytic, we can expand it in a Taylor series about a real point x as follows,

$$f(x+ih) = f(x) + ihf'(x) - h^2\frac{f''(x)}{2!} - ih^3\frac{f'''(x)}{3!} + \dots \tag{4.11}$$

Taking the imaginary parts of both sides of (4.11) and dividing the equation by h yields

$$f'(x) = \frac{\operatorname{Im}\left[f(x+ih)\right]}{h} + h^2\frac{f'''(x)}{3!} + \dots \tag{4.12}$$

We call this the *complex-step derivative approximation*. Hence the approximations is a $\mathcal{O}(h^2)$ estimate of the derivative. Like a finite-difference formula, each additional evaluation results in a column of the Jacobian (4.42), and the cost of computing the required derivatives is proportional to the number of design variables, n_x.

Because there is no subtraction operation in the complex-step derivative approximation (4.12), the only source of numerical error is the truncation error, which is $\mathcal{O}(h^2)$. By decreasing h to a small enough value, the truncation error can be made to be of the same order as the numerical precision of the evaluation of f. This constitutes a tremendous advantage over the finite-difference approaches expressed in (4.5, 4.8).

Note that if we take the real part of the Taylor series expansion (4.11), we obtain the value of the function on the real axis, i.e.,

$$f(x) = \operatorname{Re}\left[f(x+ih)\right] + h^2\frac{f''(x)}{2!} - \dots, \tag{4.13}$$

showing that the real part of the result give the value of $f(x)$ correct to $\mathcal{O}(h^2)$.

The second order errors in the function value (4.13) and the function derivative (4.12) can be eliminated when using finite-precision arithmetic by ensuring that h is sufficiently small. If ε is the relative working precision of a given algorithm, we need an h such that

$$h^2\left|\frac{f''(x)}{2!}\right| < \varepsilon\left|f(x)\right|, \tag{4.14}$$

to eliminate the truncation error of $f(x)$ in the expansion (4.13). Similarly, for the truncation error of the derivative estimate to vanish we require that

$$h^2\left|\frac{f'''(x)}{3!}\right| < \varepsilon\left|f'(x)\right|. \tag{4.15}$$

Although the step h can be set to extremely small values, it is not always possible to satisfy these conditions (4.14, 4.15), specially when $f(x), f'(x)$ tend to zero.

$$\frac{\partial F}{\partial x} \approx \frac{F(x+h) - F(x)}{h} \qquad \frac{\partial F}{\partial x} \approx \frac{\mathrm{Im}[F(x+ih)] - \mathrm{Im}[F(x)]}{\mathrm{Im}[ih]} = \frac{\mathrm{Im}[F(x+ih)]}{h}$$

Figure 4.3: Derivative approximations $\mathrm{d}F/\,\mathrm{d}x$ using a forward step in the real (left) and complex (right) axes. Here, F and x are scalars. In the complex-step method, there is no subtraction operation involved because the value of the initial point, $F(x)$, has no imaginary part.

An alternative way of deriving and understanding the complex step is to consider a function, $f = u + iv$, of the complex variable, $z = x + iy$. If f is analytic the Cauchy–Riemann equations apply, i.e.,

$$\frac{\partial u}{\partial x} = \frac{\partial v}{\partial y} \tag{4.16}$$

$$\frac{\partial u}{\partial y} = -\frac{\partial v}{\partial x}. \tag{4.17}$$

These equations establish the exact relationship between the real and imaginary parts of the function. We can use the definition of a derivative in the right hand side of the first Cauchy–Riemann Eq. (4.16) to obtain,

$$\frac{\partial u}{\partial x} = \lim_{h \to 0} \frac{v(x + i(y + h)) - v(x + iy)}{h}. \tag{4.18}$$

where h is a small real number. Since the functions that we are interested in are real functions of a real variable, we restrict ourselves to the real axis, in which case $y = 0$, $u(x) = f(x)$ and $v(x) = 0$. Eq. (4.18) can then be re-written as,

$$\frac{\partial f}{\partial x} = \lim_{h \to 0} \frac{\mathrm{Im}\left[f\left(x + ih\right)\right]}{h}. \tag{4.19}$$

For a small discrete h, this can be approximated by,

$$\frac{\partial f}{\partial x} \approx \frac{\mathrm{Im}\left[f\left(x + ih\right)\right]}{h}. \tag{4.20}$$

The property that was used to derive this approximation is a direct consequence of the analyticity of the function and it is therefore necessary that the function f be analytic in the complex plane. In a later section we will discuss to what extent a generic numerical algorithm can be considered to be an analytic function.

Fig. 4.3 illustrates the difference between the complex-step and finite-difference formulas. When using the complex-step method, the differencing quotient is evaluated using the imaginary parts of the function values and step size, and the quantity $\boldsymbol{F}(x_j)$ has no imaginary component to subtract.

Example 4.10. The Complex-Step Method Applied to a Simple Function

To show the how the complex-step method works, consider the following analytic function:

$$f(x) = \frac{\mathrm{e}^x}{\sqrt{\sin^3 x + \cos^3 x}} \tag{4.21}$$

The exact derivative at $x = 1.5$ was computed analytically to 16 digits and then compared to the results given by the complex-step (4.20) and the forward and central finite-difference approximations. The relative error in the derivative estimates given by finite-difference and the complex-step methods with the analytic result as the reference, i.e.,

$$\varepsilon = \frac{\left| f' - f'_{ref} \right|}{\left| f'_{ref} \right|}. \tag{4.22}$$

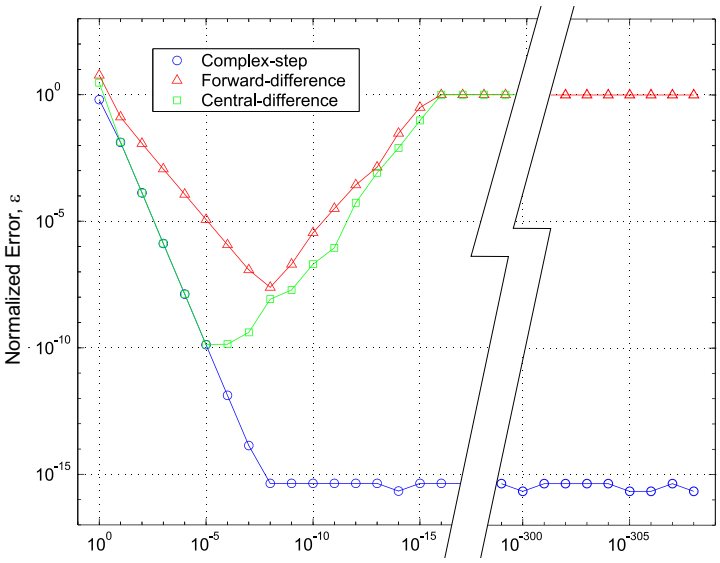


Figure 4.4: Relative error of the derivative vs. decreasing step size

As we can see in Fig. 4.4, the forward-difference estimate initially converges to the exact result at a linear rate since its truncation error is $\mathcal{O}(h)$, while the central-difference converges quadratically, as expected. However, as the step is reduced below a value of about 10^{-8} for the forward-difference and 10^{-5} for the central-difference, subtractive cancellation errors become significant and the estimates are unreliable. When the interval h is so small that no difference exists in the output (for steps smaller than 10^{-16}) the finite-difference estimates eventually yields zero and then $\varepsilon = 1$.

The complex-step estimate converges quadratically with decreasing step size, as predicted by the truncation error estimate. The estimate is practically insensitive to small step sizes and below an h of the order of 10^{-8} it achieves the accuracy of the function evaluation. Comparing the best accuracy of each of these approaches, we can see that by using finite-difference we only achieve a fraction of the accuracy that is obtained by using the complex-step approximation.

The complex-step size can be made extremely small. However, there is a lower limit on the step size when using finite precision arithmetic. The range of real numbers that can be handled in numerical computing is dependent on the particular compiler that is used. In this case, the smallest non-zero number that can be represented is 10^{-308}. If a number falls below this value, underflow occurs and the number drops to zero. Note that the estimate is still accurate down to a step of the order of 10^{-307}. Below this, underflow occurs and the estimate results in `NaN`.

Comparing the accuracy of complex and real computations, there is an increased error in basic arithmetic operations when using complex numbers, more specifically when dividing and multiplying.

In the derivation of the complex-step derivative approximation for a function f (4.20) we assume that f is analytic, i.e. that the Cauchy–Riemann equations (Eqs. (4.16) and (4.17)) apply. It is therefore important to determine to what extent this assumption holds when the value of the function is calculated by a numerical algorithm. In addition, it is useful to explain how real functions and operators can be defined such that the complex-step derivative approximation yields the correct result when used in a computer program.

Relational logic operators such as "greater than" and "less than"are usually not defined for complex numbers. These operators are often used in programs, in conjunction with conditional statements to redirect the execution thread. The original algorithm and its "complexified" version must obviously follow the same execution thread. Therefore, defining these operators to compare only the real parts of the arguments is the correct approach. Functions that choose one argument such as the maximum or the minimum values are based on relational operators. Therefore, following the previous argument, we should also choose a number based on its real part alone.

Any algorithm that uses conditional statements is likely to be a discontinuous function of its inputs. Either the function value itself is discontinuous or the discontinuity is in the first or higher derivatives. When using a finite-difference method, the derivative estimate is incorrect if the two function evaluations are within h of the discontinuity location. However, when using the complex-step method, the resulting derivative estimate is correct right up to the discontinuity. At the discontinuity, a derivative does not exist by definition, but if the function is continuous up to that point, the approximation still returns a value corresponding to the one-sided derivative. The same is true for points where a given function has singularities.

Arithmetic functions and operators include addition, multiplication, and trigonometric functions, to name only a few. Most of these have a standard complex definition that is analytic, in which case the complex-step derivative approximation yields the correct result.

The only standard complex function definition that is non-analytic is the absolute value function. When the argument of this function is a complex number, the function returns the positive real number, $|z| = \sqrt{x^2 + y^2}$. The definition of this function is not derived by imposing analyticity and therefore it does not yield the correct derivative when using the complex-step estimate. In order to derive an analytic definition of the absolute value function we must ensure that the Cauchy–Riemann equations (4.16, 4.17) are satisfied. Since we know the exact derivative of the absolute value function, we can write

$$\frac{\partial u}{\partial x} = \frac{\partial v}{\partial y} = \begin{cases} -1, & \text{if} \quad x < 0, \\ +1, & \text{if} \quad x > 0. \end{cases} \tag{4.23}$$

From Eq. (4.17), since $\partial v/\partial x = 0$ on the real axis, we get that $\partial u/\partial y = 0$ on the same axis, and therefore the real part of the result must be independent of the imaginary part of the variable. Therefore, the new sign of the imaginary part depends only on the sign of the real part of the complex number, and an analytic "absolute value" function can be defined as

$$\texttt{abs}(x + iy) = \begin{cases} -x - iy, & \text{if} \quad x < 0, \\ +x + iy, & \text{if} \quad x > 0. \end{cases} \tag{4.24}$$

Note that this is not analytic at $x = 0$ since a derivative does not exist for the real absolute value. In practice, the $x > 0$ condition is substituted by $x \geq 0$ so that we can obtain a function value for $x = 0$ and calculate the correct right-hand-side derivative at that point.

4.4.3 Implementation Procedure

The general procedure for the implementation of the complex-step method for an arbitrary computer program can be summarized as follows:

1. Substitute all `real` type variable declarations with `complex` declarations. It is not strictly necessary to declare *all* variables complex, but it is much easier to do so.

2. Define all functions and operators that are not defined for complex arguments.

3. Add a small complex step (e.g. $h = 1 \times 10^{-20}$) to the desired x, run the algorithm that evaluates f, and then compute $\partial f/\partial x$ using Eq. (4.20).

The above procedure is independent of the programming language. We now describe the details of our Fortran and C/C++ implementations.

Fortran

In Fortran 90, intrinsic functions and operators (including comparison operators) can be overloaded. This means that if a particular function or operator does not accept complex arguments, one can extend it by writing another definition that does. This feature makes it much easier to implement the complex-step method since once we overload the functions and operators, there is no need to change the function calls or conditional statements.

The complex function and operators needed for implementing the complex-step method are defined in a `complexify` Fortran 90 module. The module can be used by any subroutine in a program, including Fortran 77 subroutines and it redefines all intrinsic complex functions Operators — like for example addition and multiplication — that are intrinsically defined for complex arguments cannot be redefined, according to the Fortran 90 standard. However, the intrinsic complex definitions are suitable for our purposes.

In order to automate the implementation, we developed a script that processes Fortran source files automatically. The script inserts a statement that ensures that the complex functions module is used in every subroutine, substitutes all the real type declarations by complex ones and adds `implicit complex` statements when appropriate. The script is written in Python [35] and supports a wide range of platforms and compilers. It is also compatible with the message-passing interface (MPI) standard for parallel computing. For MPI-based code, the script converts all real types in MPI calls to complex ones. In addition, the script can also take care of file input and output by changing `format` statements. Alternatively, a separate script can be used to automatically change the input files themselves. The latest versions of both the script and the Fortran 90 module are available from a dedicated web page.[1]

This tool for implementing the complex-step method represents, in our opinion, a good compromise between ease of implementation and algorithmic efficiency. While pure algorithmic differentiation is numerically more efficient, the complex-step method requires far fewer changes to the original source code, due to the fact that complex variables are a Fortran intrinsic type. The end result is improved maintainability. Furthermore, practically all the changes are performed automatically by the use of the script.

[1] http://mdolab.engin.umich.edu/content/guide-complex-step-derivative-approximation-0

C/C++

The C/C++ implementations of the complex-step method and algorithmic differentiation are much more straightforward than in Fortran. Two different C/C++ implementations are described in this chapter.

The first procedure is analogous to the Fortran implementation, i.e., it uses complex variables and overloaded complex functions and operators. An include file, `complexify.h`, defines a new variable type called `cmplx` and all the functions that are necessary for the complex-step method. The inclusion of this file and the replacement of `double` or `float` declarations with `cmplx` is nearly all that is required.

The remaining work involves dealing with input and output routines. The usual casting of the inputs to `cmplx` and printing of the outputs using the `real()` and `imag()` functions works well. For ASCII files or terminal input and output, the use of the C++ iostream library is also possible. In this case, the "$\gg$" operator automatically reads a real number and properly casts it to `cmplx`, or reads in a complex number in the Fortran parenthesis format, e.g., "`(2.3,1.e-20)`". The "$\ll$" operator outputs in the same format.

The second method is a version of the forward mode of algorithmic differentiation. The method can be implemented by including a file called `derivify.h` and by replacing declarations of type `double` with declarations of type `surreal`. The `derivify.h` file redefines all relational operators, the basic arithmetic formulas, trigonometric functions, and other formulas in the math library when applied to the `surreal` variables. These variables contain "value" and "derivative" parts analogous to the real and imaginary parts in the complex-step method. This works just as the complex-step version, except that the step size may be set to unity since there is no truncation error.

One feature available to the C++ programmer is worth mentioning: templates. Templates make it possible to write source code that is independent of variable type declarations. This approach involves considerable work with complicated syntax in function declarations and requires some object-oriented programming. There is no need, however, to modify the function bodies themselves or to change the flow of execution, even for pure C programs. The distinct advantage is that variable types can be decided at run time, so the very same executable can run either the real-valued, the complex or the algorithmic differentiation version. This simplifies version control and debugging considerably since the source code is the same for all three versions of the program.

Other Programming Languages

In addition to the Fortran and C/C++ implementations described above, there was some experimentation with other programming languages.

Matlab: As in the case of Fortran, one must redefine functions such as `abs`, `max` and `min`. All differentiable functions are defined for complex variables. The results shown in Fig. 4.4 are actually computed using Matlab. The standard transpose operation represented by an apostrophe (`'`) poses a problem as it takes the complex conjugate of the elements of the matrix, so one should use the non-conjugate transpose represented by "dot apostrophe" (`.'`) instead.

Java: Complex arithmetic is not standardized at the moment but there are plans for its implementation. Although function overloading is possible, operator overloading is currently not supported.

Python: A simple implementation of the complex-step method for Python was also developed in this work. The `cmath` module must be imported to gain access to complex arithmetic.

Since Python supports operator overloading, it is possible to define complex functions and operators as described earlier.

Algorithmic differentiation by overloading can be implemented in any programming language that supports derived datatypes and operator overloading. For languages that do not have these features, the complex-step method can be used wherever complex arithmetic is supported.

Example 4.11. Aerostructural Design Optimization Framework

The tool that we have developed to implement the complex-step method automatically in Fortran has been tested on a variety of programs. One of the most complicated examples is a high-fidelity aerostructural solver, which is part of an MDO framework created to solve wing aerostructural design optimization problems [44, 45, 56]. The framework consists of an aerodynamic analysis and design module (which includes a geometry engine and a mesh perturbation algorithm), a linear finite-element structural solver, an aerostructural coupling procedure, and various pre-processing tools that are used to setup aerostructural design problems. The multidisciplinary nature of this solver is illustrated in Fig. 4.5 where we can see the aircraft geometry, the flow mesh and solution, and the primary structure inside the wing.

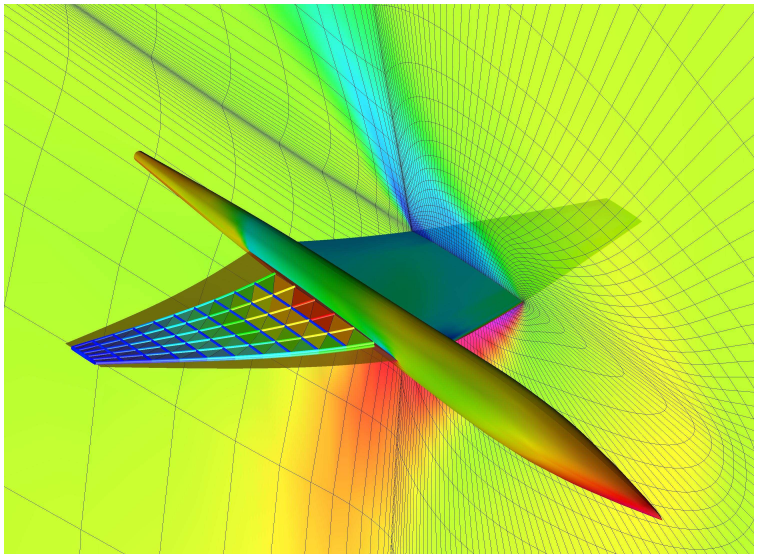


Figure 4.5: Aerostructural model and solution of a transonic jet configuration, showing a slice of the grid and the internal structure of the wing.

The aerodynamic analysis and design module, SYN107-MB [57], is a multiblock parallel flow solver for both the Euler and the Reynolds averaged Navier–Stokes equations that has been shown to be accurate and efficient for the computation of the flow around full aircraft configurations [55, 68]. An aerodynamic adjoint solver is also included in this package, enabling aerodynamic shape optimization in the absence of aerostructural interaction.

To compute the flow for this configuration, we use the CFD mesh shown in Fig. 4.5. This is a multiblock Euler mesh with 60 blocks and a total of 229,500 mesh points. The structural model of the wing consists of 6 spars, 10 ribs and skins covering the upper and lower surfaces of the wing box. A total of 640 finite elements are used in the construction of this model.

Since an analytic method for calculating derivatives has been developed for this analysis code, the complex-step method is an extremely useful reference for validation purposes [44, 45]. To validate the complex-step results for the aerostructural solver, we chose the derivative of the drag

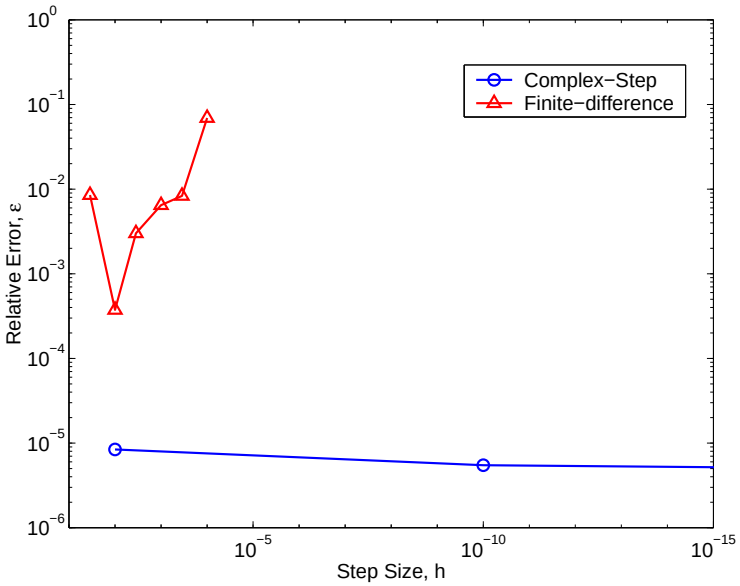


Figure 4.6: Derivative estimate vs. step size. Note that the finite-difference results are practically useless.

coefficient, C_D, with respect to a set of 18 wing shape design variables. These variables are shape perturbations in the form of Hicks–Henne functions [27], which not only control the aerodynamic shape, but also change the shape of the underlying structure.

Since the aerostructural solver is an iterative algorithm, it is useful to compare the convergence of a given function with that of its derivative, which is contained in its complex part. This comparison is shown in Fig. 4.7 for the drag coefficient and its derivative with the respect to the first shape design variable, b_1. The drag coefficient converges to the precision of the algorithm in about 300 iterations. The drag derivative converges at the same rate as the coefficient and it lags slightly, taking about 100 additional iterations to achieve the maximum precision. This is expected, since the calculation of the derivative of a given quantity is dependent on the value of that quantity [6, 22].

The minimum error in the derivative is observed to be slightly lower than the precision of the coefficient. When looking at the number of digits that are converged, the drag coefficient consistently converges to six digits, while the derivative converges to five or six digits. This small discrepancy in accuracy can be explained by the increased round-off errors of certain complex arithmetic operations such as division and multiplication due to the larger number of operations that is involved [52]. When performing multiplication, for example, the complex part is the result of two multiplications and one addition, as shown in Table 4.4. Note that this increased error does not affect the real part when such small step sizes are used.

The plot shown in Fig. 4.6 is analogous to that of Fig. 4.4, where the derivative estimates given by the complex-step and forward finite-difference methods are compared for a varying step sizes. In this case the finite-difference result has an acceptable precision only for one step size ($h = 10^{-2}$). Again, the complex-step method yields accurate results for a wide range of step sizes, from $h = 10^{-2}$ to $h = 10^{-200}$ in this case.

The results corresponding to the complete shape derivative vector are shown in Fig. 4.8. Although many different sets of finite-difference results were obtained, only the set corresponding to the optimum step is shown. The plot shows no discernible difference between the two sets of results.

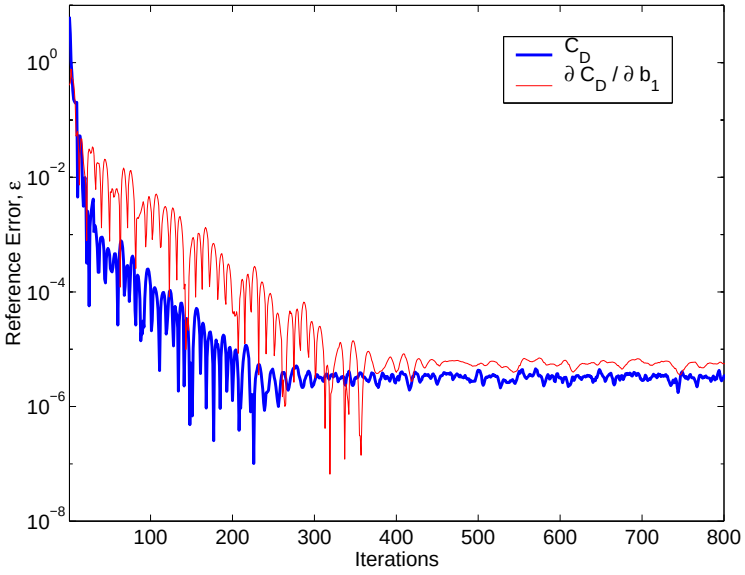


Figure 4.7: Convergence of C_D and $\partial C_D/\partial b_1$

Computation Type	Normalized Cost
aerostructural Solution	1.0
Finite difference	14.2
Complex step	34.4

Table 4.2: Normalized computational cost comparison for the calculation of the complete shape derivative vector.

Note that these derivatives account for both aerodynamic and structural effects: a variation in the shape of the wing affects the flow and the underlying structure directly. There is also an indirect effect on the flow due to the fact that the wing exhibits a new displacement field when its shape is perturbed.

A comparison of the relative computational cost of the two methods was also performed for the aerodynamic derivatives, namely for the calculation of the complete shape derivatives vector. Table 4.2 lists these costs, normalized with respect to the solution time of the aerostructural solver.

The cost of a finite-difference gradient evaluation for the 18 design variables is about 14 times the cost of a single aerostructural solution for computations that have converged to six orders of magnitude in the average density residual. Notice that one would expect this method to incur a computational cost equivalent to 19 aerostructural solutions (the solution of the baseline configuration plus one flow solution for each design variable perturbation.) The cost is lower than this value because the additional calculations start from the previously converged solution.

The cost of the complex-step procedure is more than twice of that of the finite-difference procedure since the function evaluations require complex arithmetic. We feel, however, that the complex-step calculations are worth this cost penalty since there is no need to find an acceptable step size a priori, as in the case of the finite-difference approximations.

Again, we would like to emphasize that while there was considerable effort involved in obtaining reasonable finite-difference results by optimizing the step sizes, no such effort was necessary when

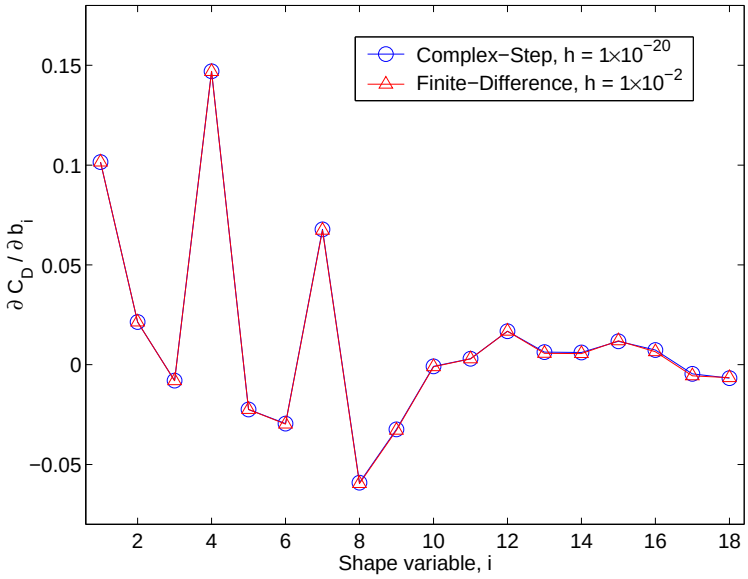


Figure 4.8: Derivative of C_D to shape functions. The finite-different results were obtained after much effort to choose the best step.

using the complex-step method.

Example 4.12. Supersonic Viscous-Inviscid Coupled Solver

This example illustrates how the complex-step method can be applied to an analysis for which it is very difficult to extract accurate finite-difference gradients. This code was developed for supporting design work of the supersonic natural laminar flow (NLF) aircraft concept [65]. It is a multidisciplinary framework, which uses input and output file manipulations to combine five computer programs including an iterative Euler solver and a boundary layer solver with transition prediction. In this framework, Python is used as the gluing language that joins the many programs. Gradients are computed with the complex-step method and with algorithmic differentiation in the Fortran, C++ and Python programming languages.

The sample derivaties was chosen to be that of the skin-friction drag coefficient with respect to the root chord. A comparison between finite-central-difference and complex-step derivatives is shown in Fig. 4.9. The plot shows the rather poor accuracy of the finite-difference derivatives for this analysis.

There are several properties of this analysis that make it difficult to extract useful finite-difference results. The most obvious is the transition from laminar to turbulent flow. It is difficult to truly smooth the movement of the transition front when transition prediction is computed on a discretized domain. Since transition has such a large effect on skin friction, this difficulty is expected to adversely affect finite-difference drag derivatives. Additionally, the discretization of the boundary layer by computing it along 12 arcs that change curvature and location as the wing planform is perturbed is suspected to cause some noise in the laminar solution as well [65].

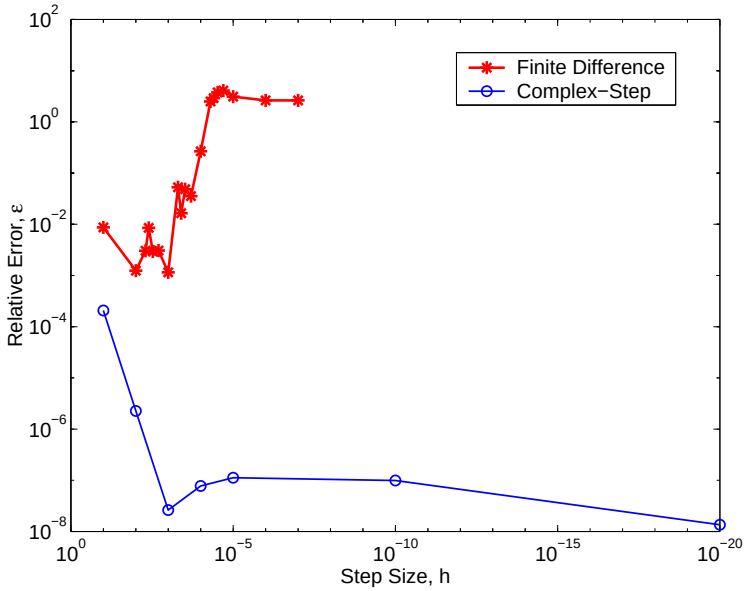


Figure 4.9: Derivative estimate vs. step size

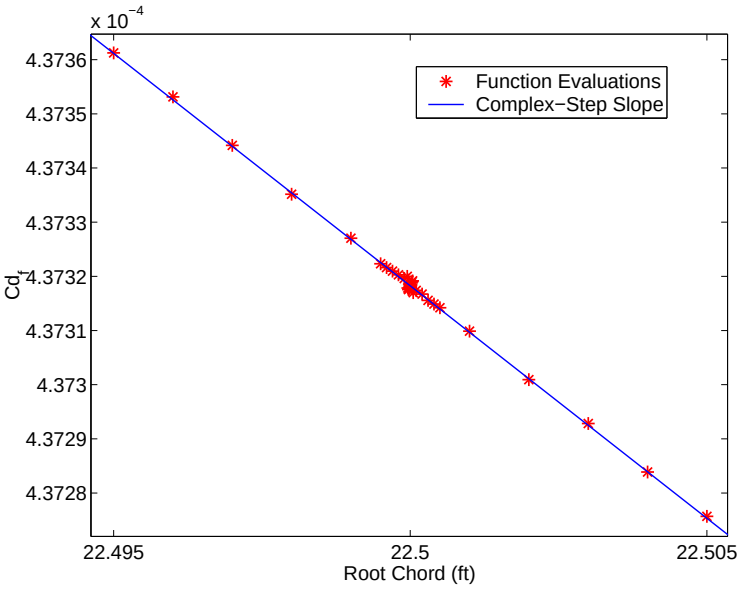


Figure 4.10: Laminar skin friction coefficient (C_D) vs. root chord

4.5 Total Derivatives of a System

To make sure the methods are explained for the most general case and show how the various methods can be derived under a single theoretical framework, it is important to characterize the computational model that is involved and precisely define the relevant terms. In the most general sense, a computational model takes a series of numerical inputs and produces outputs. As previously mentioned, the computational model is assumed to be deterministic, and it is ultimately implemented as a computer program. Depending on the type of method, we might take the computational model view or the computer program view. In either case, we sometimes refer to the model or program a *system*, since it is an interconnected series of computations.

4.5.1 Definitions

It is particularly important to realize the nested nature of the system. The most fundamental building blocks of this system are the unary and binary operations. These operations can be combined to obtain more elaborate explicit functions, which are typically expressed in one line of computer code. A more complex computation can be performed by evaluating a sequence of explicit functions V_i, where $i = 1, \ldots, n$. In its simplest form, each function in this sequence depends only on the inputs and the functions that have been computed earlier in the sequence. Thus we can represent such a computation as,

$$v_i = V_i(v_1, v_2, \ldots, v_{i-1}). \tag{4.25}$$

Here we adopt the convention that the lower case represents the *value* of a variable, and the upper case represents the *function* that computes that value. This is a distinction that will be particularly useful in developing the theory presented herein.

In the more general case, a given function might require values that have not been previously computed, i.e.,

$$v_i = V_i(v_1, v_2, \ldots, v_i, \ldots, v_n). \tag{4.26}$$

The solution of such systems require numerical methods that can be programmed by using loops where variables are updated. Numerical methods range from simple fixed-point iterations to sophisticated Newton-type algorithms. Note that loops are also used to repeat one or more computations over a computational grid.

One concept that will be used later is that it is always possible to represent any given computation without loops and dependencies — as written in Eq. (4.25) — if we *unroll* all of the loops, and represent all values a variable might take in the iteration as a separate variable that is never overwritten.

For our purposes, it is useful to generalize any computation that produces an output vector of variables $\boldsymbol{v}_{\text{out}} \subset \boldsymbol{v}$ for given an arbitrary set of input variables $\boldsymbol{v}_{\text{in}} \subset \boldsymbol{v}$. We write this computation as

$$\boldsymbol{v}_{\text{out}} = \boldsymbol{V}(\boldsymbol{v}_{\text{in}}), \tag{4.27}$$

and call it a *component*. What constitutes a component is somewhat arbitrary, but components are usually defined and organized in a way that helps us understand the overall system. The boundaries of a given component are usually determined by a number of practical factors, such as technical discipline, programming language, data dependencies or developer team.

A given component is in part characterized by the input variables it requires, and by the variables that it outputs. In the process of computing the output variables, a component might

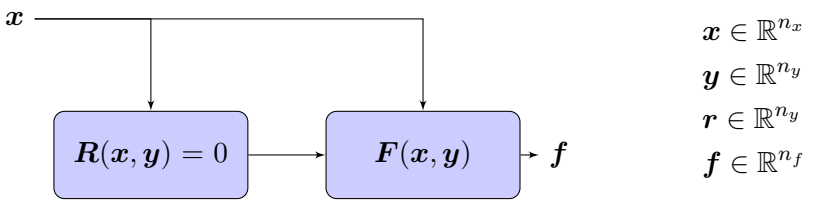


Figure 4.11: Definition of the variables involved at the solver level, showing the dependency of the quantity of interest on the design variables, both directly and through the residual equations that determine the system states

also set a number of other variables in $\boldsymbol{v}$ that are neither inputs or output, which we call *intermediate variables*.

When a component is just a series of explicit functions, we can consider the component itself to be an explicit composite function. In cases where the computation of the outputs requires iteration, it is helpful to denote the computation as a vector of *residual equations*,

$$\boldsymbol{r} = \boldsymbol{R}(\boldsymbol{v}) = 0 \tag{4.28}$$

where the algorithm changes certain components of $\boldsymbol{v}$ until all of the residuals converge to zero (or in practice, to within a small specified tolerance). The subset of $\boldsymbol{v}$ that is iterated to achieve the solution of these equations are called the *state variables*.

To relate these concepts to the usual conventions in sensitivity analysis, we now separate the subsets in $\boldsymbol{v}$ into independent variables $\boldsymbol{x}$, state variables $\boldsymbol{y}$ and quantities of interest, $\boldsymbol{f}$. Note that these do not necessary correspond exactly to the component inputs, intermediate variables and outputs, respectively. Using this notation, we can write the residual equations as,

$$\boldsymbol{r} = \boldsymbol{R}(\boldsymbol{x}, \boldsymbol{y}(\boldsymbol{x})) = 0 \tag{4.29}$$

where $\boldsymbol{y}(\boldsymbol{x})$ denotes the fact that $\boldsymbol{y}$ depends *implicitly* on $\boldsymbol{x}$ through the solution of the residual equations (4.29). It is the solution of these equations that completely determines $\boldsymbol{y}$ for a given $\boldsymbol{x}$. The functions of interest (usually included in the set of component outputs) also have the same type of variable dependence in the general case, i.e.,

$$\boldsymbol{f} = \boldsymbol{F}(\boldsymbol{x}, \boldsymbol{y}(\boldsymbol{x})). \tag{4.30}$$

When we compute the values $\boldsymbol{f}$, we assume that the state variables $\boldsymbol{y}$ have already been determined by the solution of the residual equations (4.29). The dependencies involved in the computation of the functions of interest are represented in Fig. 4.11. For our purposes, we are ultimately interested in the total derivatives of quantities $\boldsymbol{f}$ with respect to $\boldsymbol{x}$.

4.5.2 The Unifying Chain Rule

In this section, we present a single equation that unifies the methods for computing total derivatives. The methods differ in the extent to which they decompose a system, but they all come from a basic principle: a generalized chain rule.

To arrive at this form of chain rule, we start from the sequence of variables $(v_1, \ldots, v_n)$, whose values are functions of earlier variables, $v_i = V_i(v_1, \ldots, v_{i-1})$. For brevity, $V_i(v_1, \ldots, v_{i-1})$ is written

as $V_i(\cdot)$. We define a partial derivative, $\partial V_i/\partial v_j$, of a function V_i with respect to a variable v_j as

$$\frac{\partial V_i}{\partial v_j} = \frac{V_i(v_1, \ldots, v_{j-1}, v_j + h, v_{j+1}, \ldots, v_{i-1}) - V_i(\cdot)}{h}. \tag{4.31}$$

Let us consider a total variation v_i due to a perturbation v_j, which can be computed by using the sum of partial derivatives,

$$v_i = \sum_{k=j}^{i-1} \frac{\partial V_i}{\partial v_k} v_k \tag{4.32}$$

where all intermediate v's between j and i are computed and used. The total derivative is defined as,

$$\frac{\mathrm{d}v_i}{\mathrm{d}v_j} = \frac{v_i}{v_j}, \tag{4.33}$$

Using the two equations above, we can derive the following equation:

$$\frac{\mathrm{d}v_i}{\mathrm{d}v_j} = \delta_{ij} + \sum_{k=j}^{i-1} \frac{\partial V_i}{\partial v_k} \frac{\mathrm{d}v_k}{\mathrm{d}v_j}, \tag{4.34}$$

which expresses a total derivative in terms of the other total derivatives and the Jacobian of partial derivatives. The δ_{ij} term is added to account for the case in which $i = j$. Eq. (4.34) represents the chain rule for a system whose variables are $\boldsymbol{v}$.

To get a better understanding of the structure of the chain rule (4.34), and the options for performing the computation it represents, we now write it in matrix form. We can write the partial derivatives of the elementary functions V_i with respect to v_i as the square $n \times n$ Jacobian matrix,

$$\boldsymbol{D_V} = \frac{\partial V_i}{\partial v_j} = \begin{bmatrix} 0 & \cdots & & \\ \frac{\partial V_2}{\partial v_1} & 0 & \cdots & \\ \frac{\partial V_3}{\partial v_1} & \frac{\partial V_3}{\partial v_2} & 0 & \cdots \\ \vdots & \vdots & \ddots & \ddots \\ \frac{\partial V_n}{\partial v_1} & \frac{\partial V_n}{\partial v_2} & \cdots & \frac{\partial V_n}{\partial v_{n-1}} \quad 0 \end{bmatrix}, \tag{4.35}$$

where $\boldsymbol{D}$ is a differential operator. The total derivatives of the variables v_i form another Jacobian matrix of the same size that has a unit diagonal,

$$\boldsymbol{D_v} = \frac{\mathrm{d}v_i}{\mathrm{d}v_j} = \begin{bmatrix} 1 & 0 & \cdots & & \\ \frac{\mathrm{d}v_2}{\mathrm{d}v_1} & 1 & 0 & \cdots & \\ \frac{\mathrm{d}v_3}{\mathrm{d}v_1} & \frac{\mathrm{d}v_3}{\partial v_2} & 1 & 0 & \cdots \\ \vdots & \vdots & \ddots & \ddots & \\ \frac{\mathrm{d}v_n}{\mathrm{d}v_1} & \frac{\mathrm{d}v_n}{\mathrm{d}v_2} & \cdots & \frac{\mathrm{d}v_n}{\mathrm{d}v_{n-1}} & 1 \end{bmatrix}. \tag{4.36}$$

Both of these matrices are lower triangular matrices, due to our assumption that we have unrolled all of the loops.

Using this notation, the chain rule (4.34) can be written as

$$\boldsymbol{D_v} = \boldsymbol{I} + \boldsymbol{D_V}\boldsymbol{D_v}. \tag{4.37}$$

Rearranging this, we obtain,

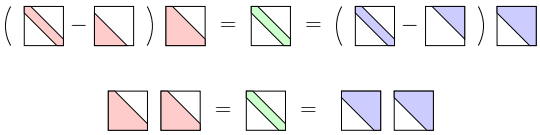
$$(\boldsymbol{I} - \boldsymbol{D_V})\,\boldsymbol{D_v} = \boldsymbol{I}. \tag{4.38}$$

where all of these matrices are square, with size $n \times n$. The matrix $(\boldsymbol{I} - \boldsymbol{D_V})$ can be formed by finding the partial derivatives, and then we can solve for the total derivatives $\boldsymbol{D_v}$. Since $(\boldsymbol{I} - \boldsymbol{D_V})$ and $\boldsymbol{D_v}$ are inverses of each other, we can further rearrange it to obtain the transposed system:

$$(\boldsymbol{I} - \boldsymbol{D_V})^T \boldsymbol{D_v}^T = \boldsymbol{I}. \tag{4.39}$$

This leads to the following symmetric relationship:

$$(\boldsymbol{I} - \boldsymbol{D_V})\, \boldsymbol{D_v} = \quad \boldsymbol{I} \quad = (\boldsymbol{I} - \boldsymbol{D_V})^T \boldsymbol{D_v}^T \tag{4.40}$$



We call the left and right hand sides of this equation the forward and reverse chain rule equations, respectively. As we will see throughout this chapter: *All methods for derivative computation can be derived from one of the forms of this chain rule* (4.40) *by changing what we mean by "variables"*, which can be seen as a level of decomposition. The various levels of decomposition are shown in Fig. 4.12 and summarized in Table 4.3.

Below Eq. (4.40) we show the structure of the matrices involved. The derivatives of interest, $\mathrm{d}\boldsymbol{f}/\,\mathrm{d}\boldsymbol{x}$, are typically the derivatives of some of the last variables with respect to some of the first variables in the sequence $(v_1, \ldots, v_n)$. They are not necessarily in sequence, but for convenience we will assume that is the case, hence,

$$\frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} = \begin{bmatrix} \dfrac{\mathrm{d}f_1}{\mathrm{d}x_1} & \cdots & \dfrac{\mathrm{d}f_1}{\mathrm{d}x_{n_x}} \\ \vdots & \ddots & \vdots \\ \dfrac{\mathrm{d}f_{n_f}}{\mathrm{d}x_1} & \cdots & \dfrac{\mathrm{d}f_{n_f}}{\mathrm{d}x_{n_x}} \end{bmatrix} = \begin{bmatrix} \dfrac{\mathrm{d}v_{(n-n_f)}}{\mathrm{d}v_1} & \cdots & \dfrac{\mathrm{d}v_{(n-n_f)}}{\mathrm{d}v_{n_x}} \\ \vdots & \ddots & \vdots \\ \dfrac{\mathrm{d}v_n}{\mathrm{d}v_1} & \cdots & \dfrac{\mathrm{d}v_n}{\mathrm{d}v_{n_x}} \end{bmatrix}, \tag{4.41}$$

This is an $n_f \times n_x$ matrix that corresponds to the lower-left block of $\boldsymbol{D_v}$ — defined in Eq. (4.36) — or the corresponding transposed upper-right block of $\boldsymbol{D_v}^T$.

Note that $\boldsymbol{D_V}$ is lower triangular, and therefore we can solve for a column of $\boldsymbol{D_v}$ using forward substitution. Conversely, $\boldsymbol{D_V}^T$ is upper triangular, and therefore we can solve for a row of $\boldsymbol{D_v}$ using back substitution. Thus, each of these versions of the chain rule incur different computational costs, depending on the shape of the Jacobian $\mathrm{d}\boldsymbol{f}/\,\mathrm{d}\boldsymbol{x}$. If $n_x > n_f$ it is advantageous to use the forward chain rule, whereas if $n_f > n_x$ the reverse chain rule is more efficient. In later sections, we will see that this basic trade-off in computational cost exists for other methods.

The two sections that follow illustrate how to use Eq. (4.40) to compute total derivatives in practice. Section **??** reviews the methods for computing the partial derivatives in the $\boldsymbol{D_V}$ matrix. Section 4.5 presents the methods for computing total derivatives such as algorithmic differentiation, analytic methods, etc., that correspond to various choices of the $\boldsymbol{v}$ — i.e. the level of decomposition of the system.

Based on the choice of $\boldsymbol{v}$ in Eq. (4.40), the $\boldsymbol{D_V}$ matrix must be assembled by finding partial derivatives of $\boldsymbol{V}$ in the appropriate context. Any intermediate variable that is used in computing v_i but is not one of the entries of $\boldsymbol{v}$ is treated as part of the black box function V_i. These intermediate variables are not held fixed while computing a partial derivative of V_i. Thus, the meaning of a partial derivative changes based on the variables, $\boldsymbol{v}$, that are exposed in the current context.

Consider a vector-valued function $\boldsymbol{F}$ with respect to a vector of independent variables $\boldsymbol{x}$. We are interested in the Jacobian,

$$\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial F_1}{\partial x_1} & \cdots & \frac{\partial F_1}{\partial x_{n_x}} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_{n_f}}{\partial x_1} & \cdots & \frac{\partial F_{n_f}}{\partial x_{n_x}} \end{bmatrix}, \tag{4.42}$$

which is an $n_f \times n_x$ matrix. In this section, we will examine three methods for computing the entries of this matrix.

Before evaluating any partial derivatives, it is necessary to make an appropriate choice of $\boldsymbol{v}$. This choice reflects the level of decomposition of the computation model, which can range between exposing only input and output variables to tracking all variables down to individual lines of code. The choice of $\boldsymbol{v}$ is the main difference between the methods for computing total derivatives. A second major difference is the technique used to solve the linear system in Eq. (4.40). In some cases, the matrix can have a triangular structure or blocks corresponding to disciplines, which may suggest different methods for solving the linear system. This is summarized in Table 4.3.

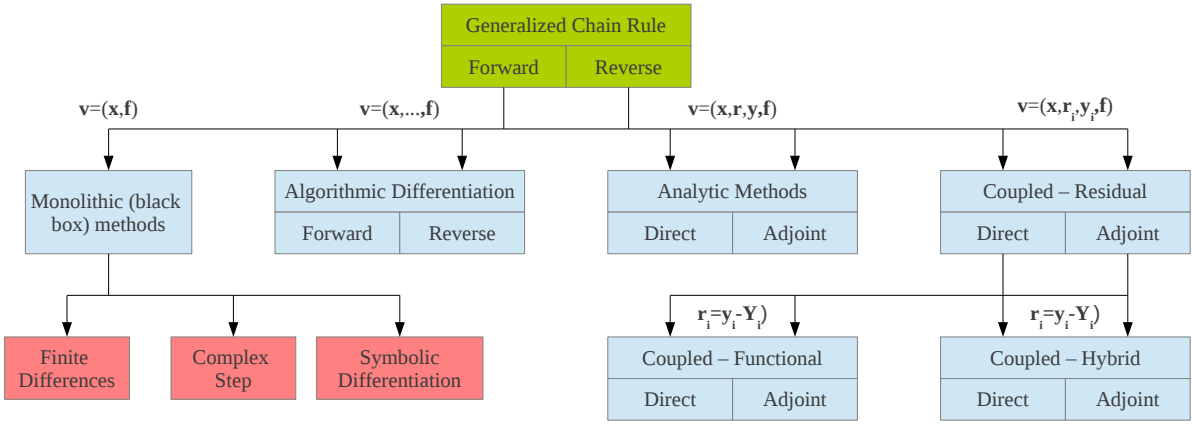


Figure 4.12: Diagram showing how the generalized chain rule unifies all derivative computation methods. Monolithic differentiation, algorithmic differentiation, analytic methods, and coupled analytic methods can all be derived from the forward or reverse mode of the generalized chain rule by making an appropriate choice of exposed variables. For black-box finite differences, complex step, and symbolic differentiation, the forward and reverse modes yield the same result. The coupled function and coupled hybrid methods can be derived from the coupled residual method by artificially defining residual and state variables and inserting these into the coupled residual equations.

Table 4.3: Classification of the methods for computing derivatives with respect to the level of decomposition, differentiation method, and strategy for solving the linear system.

	Monolithic	Analytic	Multidisciplinary Analytic	AD
Level of decomposition	Black box	Solver	Discipline	Line of code
Differentiation method	FD/CS	Any	Any	Symbolic
Solution of linear system	Trivial	Numerical	Numerical (block)	Forward substitution Back substitution

Example 4.13. Numerical Example

Before discussing the methods, we introduce a simple numerical example to which each method can be applied. It can be interpreted as an explicit function, a model with states constrained by residuals, or a multidisciplinary system, depending on the need.

This example has two inputs, $\boldsymbol{x} = [x_1, x_2]^T$, and the residual equations are,

$$\boldsymbol{R} = \begin{bmatrix} R_1(x_1, x_2, y_1, y_2) \\ R_2(x_1, x_2, y_1, y_2) \end{bmatrix} = \begin{bmatrix} x_1 y_1 + 2y_2 - \sin x_1 \\ -y_1 + x_2^2 y_2 \end{bmatrix} \tag{4.43}$$

where the $\boldsymbol{y} = [y_1 \quad y_2]^T$ are the state variables. The final piece of the model are the output functions,

$$\boldsymbol{F} = \begin{bmatrix} F_1(x_1, x_2, y_1, y_2) \\ F_2(x_1, x_2, y_1, y_2) \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \sin x_1 \end{bmatrix} \tag{4.44}$$

To drive the residuals to zero, we have to solve the following linear system,

$$\begin{bmatrix} x_1 & 2 \\ -1 & x_2^2 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \sin x_1 \\ 0 \end{bmatrix} \tag{4.45}$$

In Fig. 4.13 we list a Fortran program that takes the two input variables, solves the above linear system and computes the two output variables. In this case, the algorithm solves the system directly and there are no loops. The v's introduced above correspond to each variable assignment, i.e.,

$$\boldsymbol{v} = [\texttt{x(1)} \quad \texttt{x(2)} \quad \texttt{det} \quad \texttt{y(1)} \quad \texttt{y(2)} \quad \texttt{f(1)} \quad \texttt{f(2)}]^T \tag{4.46}$$

```
FUNCTION F(x)
  REAL :: x(2), det, y(2), f(2)
  det = 2 + x(1)*x(2)**2
  y(1) = x(2)**2*SIN(x(1))/det
  y(2) = SIN(x(1))/det
  f(1) = y(1)
  f(2) = y(2)*SIN(x(1))
  RETURN
END FUNCTION F
```

Figure 4.13: Fortran code for the numerical example computational model

The objective is to compute the derivatives of both outputs with respect to both inputs, i.e., the Jacobian,

$$\frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} = \begin{bmatrix} \frac{\mathrm{d}f_1}{\mathrm{d}x_1} & \frac{\mathrm{d}f_1}{\mathrm{d}x_2} \\ \frac{\mathrm{d}f_2}{\mathrm{d}x_1} & \frac{\mathrm{d}f_2}{\mathrm{d}x_2} \end{bmatrix} \tag{4.47}$$

We will use this example in later sections to show the application of finite differences, the complex step, algorithmic differentiation, and analytic methods.

4.6 Monolithic Differentiation

In monolithic differentiation, the entire computational model is treated as a "black box" and the only variables that are tracked are the inputs and outputs. This may be the only option in cases in which there are severe time constraints or it is difficult to get access to the source code. In this case, both the forward and reverse modes of the generalized chain rule (4.40) reduce to

$$\frac{\mathrm{d}f_i}{\mathrm{d}x_j} = \frac{\partial F_i}{\partial x_j} \tag{4.48}$$

for each input x_j and output variable f_i.

Example 4.14. Finite-Difference and Complex-Step Methods Applied to Sample Code

The monolithic approach treats the entire piece of code as a black box whose internal variables and computations are unknown. Thus, the tracked variables at this level of decomposition are

$$v_1 = x_1, \quad v_2 = x_2, \quad v_3 = f_1, \quad v_4 = f_2 \tag{4.49}$$

Inserting this choice of v_i into Eq. (4.40), both the forward and reverse chain rule equations yield in this case,

$$\frac{\mathrm{d}f_1}{\mathrm{d}x_1} = \frac{\partial f_1}{\partial x_1}, \quad \frac{\mathrm{d}f_1}{\mathrm{d}x_2} = \frac{\partial f_1}{\partial x_2}, \ldots$$

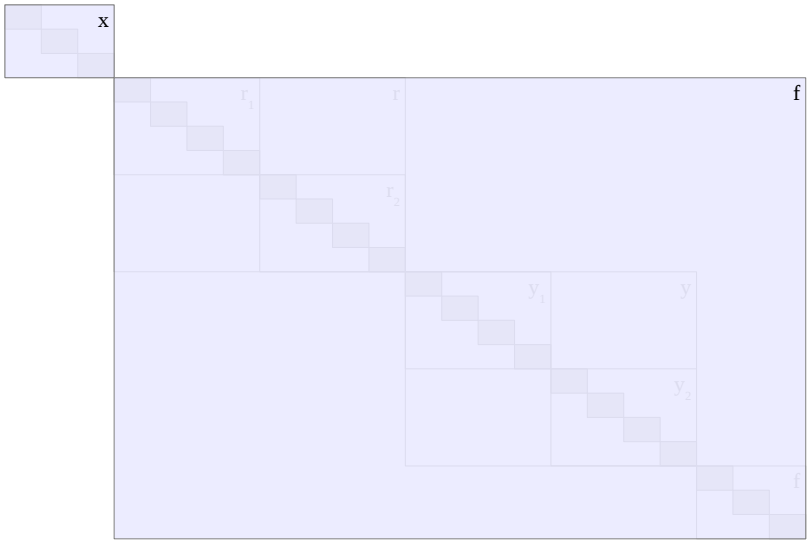
For this method, computing df_1/dx_1 simply amounts to computing $\partial f_1/\partial x_1$, which we can do by using the forward-difference formula (with step size $h = 10^{-5}$), yielding

$$\frac{\partial f_1}{\partial x_1} \approx \frac{f_1(x_1 + h, x_2) - f_1(x_1, x_2)}{h} = 0.08660230140\color{red}79,$$

or the complex-step method (with step size $h = 10^{-15}$), yielding

$$\frac{\partial f_1}{\partial x_1} \approx \frac{\mathrm{Im}\left[f_1(x_1 + ih, x_2)\right]}{h} = 0.0866039925\color{red}329.$$

The digits that agree with the exact derivative are shown in blue and those that are incorrect are in red.



$$\boldsymbol{v} = [\underbrace{v_1, \ldots, v_{n_x}}_{\boldsymbol{x}}, \underbrace{v_{(n-n_f)}, \ldots, v_n}_{\boldsymbol{f}}]^T$$

Figure 4.14: Decomposition level for monolithic differentiation: the variables $\boldsymbol{v}$ are all of the variables assigned in the computer program.

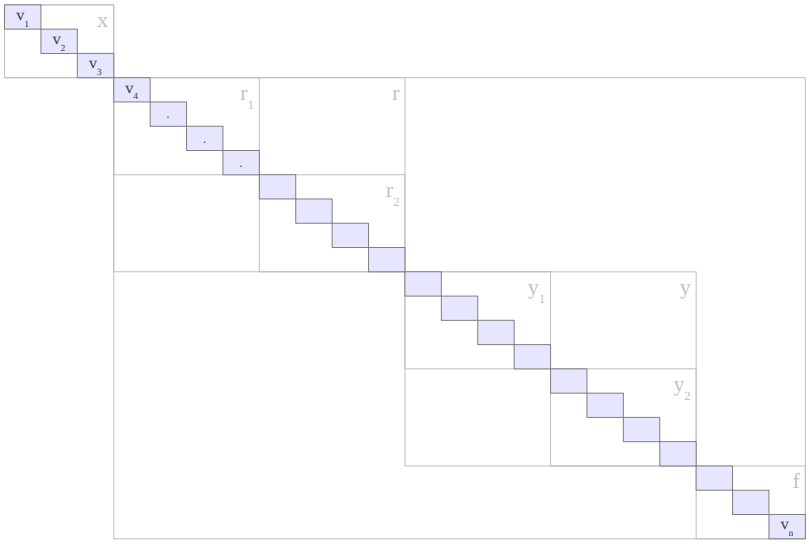
4.7 Algorithmic Differentiation

Algorithmic differentiation (AD) — also known as computational differentiation or automatic differentiation — is a well known method based on the systematic application of the differentiation chain rule to computer programs [21, 50]. Although this approach is as accurate as an analytic method, it is potentially much easier to implement since the implementation can be done automatically. To explain AD, we start by describing the basic theory and how it relates to the chain rule identity (4.40) introduced in the previous section. We then explain how the method is implemented in practice, and show an example.

From the AD perspective, the variables $\boldsymbol{v}$ in the chain rule (4.40) are all of the variables assigned in the computer program, and AD applies the chain rule for every single line in the program. The computer program thus can be considered a sequence of explicit functions V_i, where $i = 1, \ldots, n$. In its simplest form, each function in this sequence depends only on the inputs and the functions that have been computed earlier in the sequence, as expressed in the functional dependence (4.25).

Again, for this assumption to hold, we assume that all of the loops in the program are *unrolled*, and therefore no variables are overwritten and each variable only depends on earlier variables in the sequence. Later, when we explain how AD is implemented, it will become clear that this assumption is not restrictive, as programs iterate the chain rule (and thus the total derivatives) together with the program variables, converging to the correct total derivatives.

In the AD perspective, the independent variables $\boldsymbol{x}$ and the quantities of interest $\boldsymbol{f}$ are assumed to be in the vector of variables $\boldsymbol{v}$. Typically, the design variables are among the v's with lower indices, and the quantities of interest are among the last quantities. Thus, to make clear the



$$\boldsymbol{v} = [\underbrace{v_1, \ldots, v_{n_x}}_{\boldsymbol{x}}, \ldots, v_j, \ldots, v_i, \ldots, \underbrace{v_{(n-n_f)}, \ldots, v_n}_{\boldsymbol{f}}]^T$$

Figure 4.15: Decomposition level for algorithmic differentiation: the variables $\boldsymbol{v}$ are all of the variables assigned in the computer program.

connection to the other derivative computation methods, we group these variables as follows,

$$\boldsymbol{v} = [\underbrace{v_1, \ldots, v_{n_x}}_{\boldsymbol{x}}, \ldots, v_j, \ldots, v_i, \ldots, \underbrace{v_{(n-n_f)}, \ldots, v_n}_{\boldsymbol{f}}]^T. \tag{4.50}$$

The chain rule (4.34) introduced in the previous section was

$$\frac{\mathrm{d}v_i}{\mathrm{d}v_j} = \delta_{ij} + \sum_{k=j}^{i-1} \frac{\partial V_i}{\partial v_k} \frac{\mathrm{d}v_k}{\mathrm{d}v_j}, \tag{4.51}$$

where the V represent explicit functions, each defined by a single line in the computer program. The partial derivatives, $\partial V_i / \partial v_k$ can be automatically differentiated symbolically by applying another chain rule within the function defined by the respective line.

The chain rule (4.51) can be solved in two ways. In the *forward mode*, we choose one v_j and keep j fixed. Then we work our way forward in the index $i = 1, 2, \ldots, n$ until we get the desired total derivative. In the *reverse mode*, on the other hand, we fix v_i (the quantity we want to differentiate) and work our way backward in the index $j = n, n-1, \ldots, 1$ all of the way to the independent variables. We now describe these two modes in more detail, and compare the computational costs associated with each of them.

4.7.1 Forward Mode Matrix Equations

To get a better understanding of the structure of the chain rule (4.34), and the options for performing that computation, we now write it in the matrix form (4.38):

$$(\boldsymbol{I} - \boldsymbol{D_V})\, \boldsymbol{D_v} = \boldsymbol{I} \quad \Rightarrow$$

$$\begin{bmatrix} 1 & 0 & \cdots & & \\ -\frac{\partial V_2}{\partial v_1} & 1 & 0 & \cdots & \\ -\frac{\partial V_3}{\partial v_1} & -\frac{\partial V_3}{\partial v_2} & 1 & 0 & \cdots \\ \vdots & \vdots & \ddots & \ddots & \\ -\frac{\partial V_n}{\partial v_1} & -\frac{\partial V_n}{\partial v_2} & \cdots & -\frac{\partial V_n}{\partial v_{n-1}} & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & \cdots & & \\ \frac{\mathrm{d}v_2}{\mathrm{d}v_1} & 1 & 0 & \cdots & \\ \frac{\mathrm{d}v_3}{\mathrm{d}v_1} & \frac{\mathrm{d}v_3}{\partial v_2} & 1 & 0 & \cdots \\ \vdots & \vdots & \ddots & \ddots & \\ \frac{\mathrm{d}v_n}{\mathrm{d}v_1} & \frac{\mathrm{d}v_n}{\mathrm{d}v_2} & \cdots & \frac{\mathrm{d}v_n}{\mathrm{d}v_{n-1}} & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & & \\ 0 & 1 & 0 & \cdots & \\ 0 & 0 & 1 & 0 & \cdots \\ \vdots & \vdots & \vdots & \ddots & \ddots \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{4.52}$$

The terms that we ultimately want to compute are the total derivatives of quantities of interest with respect to the design variables, corresponding to a block in the $\boldsymbol{D_v}$ matrix in the lower left. Using the definition expressed in Eq. (4.42), this block is

$$\frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} = \begin{bmatrix} \frac{\mathrm{d}f_1}{\mathrm{d}x_1} & \cdots & \frac{\mathrm{d}f_1}{\mathrm{d}x_{n_x}} \\ \vdots & \ddots & \vdots \\ \frac{\mathrm{d}f_{n_f}}{\mathrm{d}x_1} & \cdots & \frac{\mathrm{d}f_{n_f}}{\mathrm{d}x_{n_x}} \end{bmatrix} = \begin{bmatrix} \frac{\mathrm{d}v_{(n-n_f)}}{\mathrm{d}v_1} & \cdots & \frac{\mathrm{d}v_{(n-n_f)}}{\mathrm{d}v_{n_x}} \\ \vdots & \ddots & \vdots \\ \frac{\mathrm{d}v_n}{\mathrm{d}v_1} & \cdots & \frac{\mathrm{d}v_n}{\mathrm{d}v_{n_x}} \end{bmatrix}, \tag{4.53}$$

which is an $n_f \times n_x$ matrix.

The forward mode is equivalent to solving the linear system (4.53) for one column of $\boldsymbol{D_v}$. Since $(\boldsymbol{I} - \boldsymbol{D_V})$ is a lower triangular matrix, this solution can be accomplished by forward substitution. In the process, we end up computing the derivative of the chosen quantity with respect to all of the other variables. The cost of this procedure is similar to the cost of the procedure that computes the v's, and as we will see in Section 19, the forward AD operations are interspersed with the operations that compute the v's in the original computer code.

Example 4.15. Forward Mode Applied to Sample Code

We now illustrate the application of automatic differentiation to the numerical example introduced in Section 13. Assuming we use the program listed in Fig. 4.13, the variables are

$$\boldsymbol{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \end{bmatrix} = \begin{bmatrix} \texttt{x(1)} \\ \texttt{x(2)} \\ \texttt{det} \\ \texttt{y(1)} \\ \texttt{y(2)} \\ \texttt{f(1)} \\ \texttt{f(2)} \end{bmatrix}. \tag{4.54}$$

We apply the forward mode chain rule (4.52) using this choice of variables and evaluate each partial derivative using symbolic differentiation to obtain

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ -v_2^2 & -2v_1v_2 & 1 & 0 & 0 & 0 & 0 \\ -\frac{v_2^2 \cos v_1}{v_3} & -\frac{2v_2 \sin v_1}{v_3} & \frac{v_2^2 \sin v_1}{v_3^2} & 1 & 0 & 0 & 0 \\ -\frac{\cos v_1}{v_3} & 0 & \frac{\sin v_1}{v_3^2} & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 & 0 \\ -v_5 \cos v_1 & 0 & 0 & 0 & -\sin v_1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \frac{\mathrm{d}v_2}{\mathrm{d}v_1} & 1 \\ \frac{\mathrm{d}v_3}{\mathrm{d}v_1} & \frac{\mathrm{d}v_3}{\partial v_2} \\ \frac{\mathrm{d}v_4}{\mathrm{d}v_1} & \frac{\mathrm{d}v_4}{\partial v_2} \\ \frac{\mathrm{d}v_5}{\mathrm{d}v_1} & \frac{\mathrm{d}v_5}{\partial v_2} \\ \frac{\mathrm{d}v_6}{\mathrm{d}v_1} & \frac{\mathrm{d}v_6}{\partial v_2} \\ \frac{\mathrm{d}v_7}{\mathrm{d}v_1} & \frac{\mathrm{d}v_7}{\partial v_2} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}. \tag{4.55}$$

In the equation above, we have only kept the first two columns of the matrices $\boldsymbol{D_v}$ and $\boldsymbol{I}$, because the derivatives of interest are,

$$\begin{bmatrix} \frac{\mathrm{d}f_1}{\mathrm{d}x_1} & \frac{\mathrm{d}f_1}{\mathrm{d}x_2} \\ \frac{\mathrm{d}f_2}{\mathrm{d}x_1} & \frac{\mathrm{d}f_2}{\mathrm{d}x_2} \end{bmatrix} = \begin{bmatrix} \frac{\mathrm{d}v_6}{\mathrm{d}v_1} & \frac{\mathrm{d}v_6}{\mathrm{d}v_2} \\ \frac{\mathrm{d}v_7}{\mathrm{d}v_1} & \frac{\mathrm{d}v_7}{\mathrm{d}v_2} \end{bmatrix}, \tag{4.56}$$

which correspond to the bottom left-most 2×2 block in the $\boldsymbol{D_v}$ matrix. After inserting the values of v_i into the 7×7 matrix, these four derivatives of interested can be computed by performing two forward substitutions to obtain the first two columns of $\boldsymbol{D_v}$. In addition to computing the derivatives of interest, we have also computed the derivatives of all variables with respect to v_1 and v_2. In Sections 4.7.3 and Example 19 we will see how this is implemented in practice.

4.7.2 Reverse Mode Matrix Equations

The matrix representation for the reverse mode of algorithmic differentiation is given by Eq. (4.39), which expands to,

$$\left(\boldsymbol{I} - \boldsymbol{D_V}\right)^T \boldsymbol{D_v}^T = \boldsymbol{I} \quad \Rightarrow$$

$$\begin{bmatrix} 1 & -\frac{\partial V_2}{\partial v_1} & -\frac{\partial V_3}{\partial v_1} & \cdots & -\frac{\partial V_n}{\partial v_1} \\ 0 & 1 & -\frac{\partial V_3}{\partial v_2} & \cdots & -\frac{\partial V_n}{\partial v_2} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & 1 & -\frac{\partial V_n}{\partial v_{n-1}} \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & \frac{\mathrm{d}v_2}{\mathrm{d}v_1} & \frac{\mathrm{d}v_3}{\mathrm{d}v_1} & \cdots & \frac{\mathrm{d}v_n}{\mathrm{d}v_1} \\ 0 & 1 & \frac{\mathrm{d}v_3}{\mathrm{d}v_2} & \cdots & \frac{\mathrm{d}v_n}{\mathrm{d}v_2} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & 1 & \frac{\mathrm{d}v_n}{\mathrm{d}v_{n-1}} \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & & \\ 0 & 1 & 0 & \cdots & \\ 0 & 0 & 1 & 0 & \cdots \\ \vdots & \vdots & \vdots & \ddots & \ddots \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{4.57}$$

The block matrix we want to compute is in the upper right section of $\boldsymbol{D_v}^T$ and now its size is $n_x \times n_f$. As with the forward mode, we need to solve this linear system one column at the time, but now each column yields the derivatives of the chosen quantity with respect to all the other variables. Because the matrix $(\boldsymbol{I} - \boldsymbol{DV})^T$ is upper triangular, the system can be solved using back substitution.

Example 4.16. Reverse Mode Applied to Sample Code

We now apply the reverse form of the chain rule (4.57) to our example. Again, since we are only interested in the derivatives of the outputs with respect to the inputs, we can eliminate some of the columns of $\boldsymbol{D_v}^T$. In this case we only want to keep the last two columns, so we obtain

$$\begin{bmatrix} 1 & 0 & -v_2^2 & -\frac{v_2^2 \cos v_1}{v_3} & -\frac{\cos v_1}{v_3} & 0 & -v_5 \cos v_1 \\ 0 & 1 & -2v_1 v_2 & -\frac{2v_2 \sin v_1}{v_3} & 0 & 0 & 0 \\ 0 & 0 & 1 & \frac{v_2^2 \sin v_1}{v_3^2} & \frac{\sin v_1}{v_3^2} & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & -\sin v_1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{\mathrm{d}v_6}{\mathrm{d}v_1} & \frac{\mathrm{d}v_7}{\partial v_1} \\ \frac{\mathrm{d}v_6}{\mathrm{d}v_2} & \frac{\mathrm{d}v_7}{\partial v_2} \\ \frac{\mathrm{d}v_6}{\mathrm{d}v_3} & \frac{\mathrm{d}v_7}{\partial v_3} \\ \frac{\mathrm{d}v_6}{\mathrm{d}v_4} & \frac{\mathrm{d}v_7}{\partial v_4} \\ \frac{\mathrm{d}v_6}{\mathrm{d}v_5} & \frac{\mathrm{d}v_7}{\partial v_5} \\ 1 & \frac{\mathrm{d}v_7}{\mathrm{d}v_6} \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{4.58}$$

where the derivatives of interest are the top 2×2 block in the $\boldsymbol{D_v}$ matrix. In contrast to the forward mode, the derivatives of interest are computed by performing two back substitutions, through which the derivatives of v_6 and v_7 with respect to all variables are incidentally computed.

Example 4.17. Forward mode differentiation of a simple function by hand

Sometimes it is easier to understand the details by working out a simple function by hand. The function we will use has two outputs

$$\begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} (x_1 x_2 + \sin x_1)\left(3x_2^2 + 6\right) \\ x_1 x_2 + x_2^2 \end{bmatrix} \tag{4.59}$$

and is evaluated at $x = [\pi/4, 2]$.

The solution can be obtained by hand differentiation,

$$\frac{\partial f}{\partial x} = \begin{bmatrix} (x_2 + \cos x_1)\left(3x_2^2 + 6\right) & x_1\left(3x_2^2 + 6\right) + 6x_2\left(x_1 x_2 + \sin x_1\right) \\ x_2 & x_1 + 2x_2 \end{bmatrix} \tag{4.60}$$

This matrix of sensitivities at the specified point is,

$$\frac{\partial f}{\partial x} = \begin{bmatrix} 48.73 & 41.47 \\ 2.00 & 4.79 \end{bmatrix} \tag{4.61}$$

However, the point of this example is to show how the chain rule can be systematically applied in an automated fashion. To illustrate this more clearly, we write the function as a series of unary and binary computations:

$$\begin{aligned}
t_1 &= x_1 \\
t_2 &= x_2 \\
t_3 &= T_3(t_1) = \sin t_1 \\
t_4 &= T_4(t_1, t_2) = t_1 t_2 \\
t_5 &= T_5(t_2) = t_2^2 \\
t_6 &= 3 \\
t_7 &= T_7(t_3, t_4) = t_3 + t_4 \\
t_8 &= T_8(t_5, t_6) = t_5 t_6 \\
t_9 &= 6 \\
t_{10} &= T_{10}(t_8, t_9) = t_8 + t_9 \\
t_{11} &= T_{11}(t_7, t_{10}) = t_7 t_{10} \quad (= f_1) \\
t_{12} &= T_{12}(t_4, t_5) = t_4 + t_5 \quad (= f_2)
\end{aligned}$$

Thus in this case, $m = 12$. A graph of the dependencies can be see in Fig. 4.16

Now let's use the forward mode to compute $\partial f_1/\partial x_1$ in our example, which is $\partial t_{11}/\partial t_1$. Thus, we set $j = 1$ and keep it fixed, and then vary $i = 1, 2, \ldots, 11$. Note that in the sum in the chain

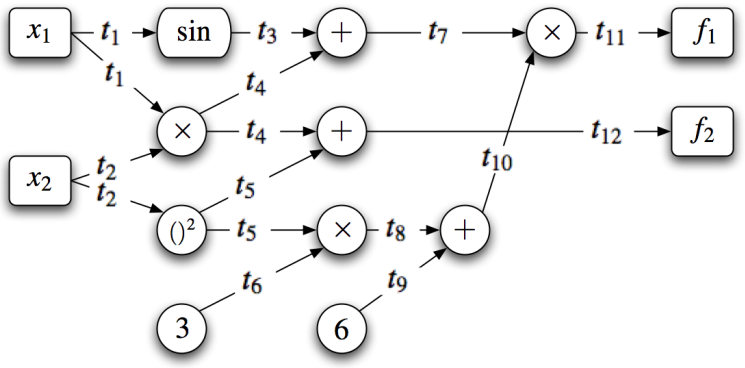


Figure 4.16: Graph showing dependency of independent, intermediate and dependent variables.

rule, we only include the k's for which $\partial T_i/\partial t_k \neq 0$.

$$\frac{\partial t_1}{\partial t_1} = 1$$
$$\frac{\partial t_2}{\partial t_1} = 0$$
$$\frac{\partial t_3}{\partial t_1} = \frac{\partial T_3}{\partial t_1}\frac{\partial t_1}{\partial t_1} = \cos t_1 \times 1 = \cos t_1$$
$$\frac{\partial t_4}{\partial t_1} = \frac{\partial T_4}{\partial t_1}\frac{\partial t_1}{\partial t_1} + \frac{\partial T_4}{\partial t_2}\frac{\partial t_2}{\partial t_1} = t_2 \times 1 + t_1 \times 0 = t_2$$
$$\frac{\partial t_5}{\partial t_1} = \frac{\partial T_5}{\partial t_2}\frac{\partial t_2}{\partial t_1} = 2t_2 \times 0 = 0$$
$$\frac{\partial t_6}{\partial t_1} = 0$$
$$\frac{\partial t_7}{\partial t_1} = \frac{\partial T_7}{\partial t_3}\frac{\partial t_3}{\partial t_1} + \frac{\partial T_7}{\partial t_4}\frac{\partial t_4}{\partial t_1} = 1 \times \cos t_1 + 1 \times t_2 = \cos t_1 + t_2$$
$$\frac{\partial t_8}{\partial t_1} = \frac{\partial T_8}{\partial t_5}\frac{\partial t_5}{\partial t_1} + \frac{\partial T_8}{\partial t_6}\frac{\partial t_6}{\partial t_1} = t_6 \times 0 + t_5 \times 0 = 0$$
$$\frac{\partial t_9}{\partial t_1} = 0$$
$$\frac{\partial t_{10}}{\partial t_1} = \frac{\partial T_{10}}{\partial t_8}\frac{\partial t_8}{\partial t_1} + \frac{\partial T_{10}}{\partial t_9}\frac{\partial t_9}{\partial t_1} = 1 \times 0 + 1 \times 0 = 0$$
$$\frac{\partial t_{11}}{\partial t_1} = \frac{\partial T_{11}}{\partial t_7}\frac{\partial t_7}{\partial t_1} + \frac{\partial T_{11}}{\partial t_{10}}\frac{\partial t_{10}}{\partial t_1} = t_{10}\left(\cos t_1 + t_2\right) + t_7 \times 0 = \left(3t_2^2 + 6\right)\left(\cos t_1 + t_2\right)$$

Note that although we did not set out to compute $\partial t_{12}/\partial t_1$, this can be found with little additional cost:

$$\frac{\partial t_{12}}{\partial t_1} = \frac{\partial T_{12}}{\partial t_4}\frac{\partial t_4}{\partial t_1} + \frac{\partial T_{12}}{\partial t_5}\frac{\partial t_5}{\partial t_1} = 1 \times t_2 + 1 \times 0 = t_2$$

Example 4.18. Reverse mode differentiation of a simple function by hand

For the reverse mode, it is useful to have in mind the dependence of all the intermediate variables. This is given by the graph of the algorithm, shown in Figure 4.16 for the case of our simple function.

To use the chain rule in reverse, we set $i = 11$ and loop $j = 11, 10, \ldots, 1$:

$$\frac{\partial t_{11}}{\partial t_{11}} = 1$$
$$\frac{\partial t_{11}}{\partial t_{10}} = \frac{\partial T_{11}}{\partial t_{10}}\frac{\partial t_{10}}{\partial t_{10}} = t_7 \times 1 = t_7$$
$$\frac{\partial t_{11}}{\partial t_9} = \frac{\partial T_{11}}{\partial t_9}\frac{\partial t_9}{\partial t_9} + \frac{\partial T_{11}}{\partial t_{10}}\frac{\partial t_{10}}{\partial t_9} = 0 \times 1 + t_7 \times 1 = t_7$$
$$\frac{\partial t_{11}}{\partial t_8} = \frac{\partial T_{11}}{\partial t_8}\frac{\partial t_8}{\partial t_8} + \frac{\partial T_{11}}{\partial t_9}\frac{\partial t_9}{\partial t_8} + \frac{\partial T_{11}}{\partial t_{10}}\frac{\partial t_{10}}{\partial t_8} = 0 \times 1 + 0 \times 0 + t_7 \times 1 = t_7$$
$$\frac{\partial t_{11}}{\partial t_7} = \frac{\partial T_{11}}{\partial t_7}\frac{\partial t_7}{\partial t_7} + \ldots + \frac{\partial T_{11}}{\partial t_{10}}\frac{\partial t_{10}}{\partial t_7} = t_{10} \times 1 + t_7 \times 0 = t_{10}$$
$$\frac{\partial t_{11}}{\partial t_6} = t_5 t_7$$
$$\frac{\partial t_{11}}{\partial t_5} = t_7 t_6$$
$$\frac{\partial t_{11}}{\partial t_4} = t_{10}$$
$$\frac{\partial t_{11}}{\partial t_3} = t_{10}$$
$$\frac{\partial t_{11}}{\partial t_2} = t_{10}t_1 + t_7 t_6 2t_2 = \left(3t_2^2 + 6\right)t_1 + 6t_2\left(\sin t_1 + t_1 t_2\right)$$
$$\frac{\partial t_{11}}{\partial t_1} = t_{10}\cos t_1 + t_{10}t_2 = \left(3x_2^2 + 6\right)\left(\cos x_1 + x_2\right)$$

Note that although we didn't set out to compute $\partial t_{11}\partial t_2$, it had to be computed anyway.

4.7.3 Implementation and Tools

There are two main ways of implementing AD: by source code transformation or by using derived datatypes and operator overloading.

To implement AD by source transformation, the whole source code must be processed with a parser and all the derivative calculations are introduced as additional lines of code. This approach and is demonstrated in Example 19. The resulting source code for large programs is greatly enlarged and it becomes practically unreadable.This fact might constitute an implementation disadvantage as it becomes impractical to debug this new extended code. One has to work with the original source, and every time it is changed (or if different derivatives are desired) one must rerun the parser before compiling a new version. The advantage is that this method tends to yield faster code.

In order to use derived types and operator overloading, we need a language that support this feature, such as Fortran 90 or C++ [21, 54]. To implement automatic differentiation using this feature, a new type of data structure is created that contains both the value and its derivative. Hence each real number v is replaced by a type that includes not only the original real number, but the corresponding derivative as well, i.e., $\bar{v} = (v, \mathrm{d}v)$. Then, all operations are redefined (overloaded) such that, in addition to the result of the original operations, they yield the derivative of that operation as well [21].

There are AD tools available for a most programming languages, including Fortran, C/C++[8, 11, 23], and Matlab. They have been extensively developed and provide the user with great functionality, including the calculation of higher-order derivatives and reverse mode options. ADIFOR [12],

TAF [17], TAMC [20] and Tapenade [24, 53] are some of the tools available for Fortran that use source transformation. The necessary changes to the source code are made automatically. The operator overloading approach is used in the following tools: AD01 [54], ADOL-F [62], IMAS [58] and OPTIMA90. Although it is in theory possible to have a script make the necessary changes in the source code automatically, none of these tools have this facility and the changes must be done manually.

One significant connection to make is that the complex-step method is equivalent to the forward mode of AD with an operator overloading implementation, as explained by Martins et al. [46]. When using the forward mode, for each intermediate variable in the algorithm, a variation due to one input variable is carried through. This is very similar to the way the complex-step method works. To illustrate this, suppose we want to differentiate the multiplication operation, $f = x_1 x_2$, with respect to x_1. Table 4.4 compares how the differentiation would be performed using either automatic differentiation or the complex-step method.

Automatic	Complex-Step
$\Delta x_1 = 1$	$h_1 = 10^{-20}$
$\Delta x_2 = 0$	$h_2 = 0$
$f = x_1 x_2$	$f = (x_1 + ih_1)(x_2 + ih_2)$
$\Delta f = x_1 \Delta x_2 + x_2 \Delta x_1$	$f = x_1 x_2 - h_1 h_2 + i(x_1 h_2 + x_2 h_1)$
$\mathrm{d}f/dx_1 = \Delta f$	$\mathrm{d}f/dx_1 = \mathrm{Im}\, f/h$

Table 4.4: The differentiation of the multiplication operation $f = x_1 x_2$ with respect to x_1 using automatic differentiation and the complex-step derivative approximation.

As we can see, automatic differentiation stores the derivative value in a separate set of variables while the complex step carries the derivative information in the imaginary part of the variables. It is shown that in this case, the complex-step method performs one additional operation — the calculation of the term $h_1 h_2$ — which, for the purposes of calculating the derivative is superfluous (and equals zero in this particular case). The complex-step method will nearly always include these unnecessary computations which correspond to the higher order terms in the Taylor series expansion. For very small h, when using finite precision arithmetic, these terms have no effect on the real part of the result.

Although this example involves only one operation, both methods work for an algorithm involving an arbitrary sequence of operations by propagating the variation of one input forward throughout the code. This means that in order to calculate n derivatives, the differentiated code must be executed n times.

Example 4.19. Algorithmic Differentiation Using Source Code Transformation

In practice, automatic differentiation performs the sequence of operations describe in Sections 4.7.1 and 4.7.2 by changing the original source code. Applying the forward mode of the source code transformation tool Tapenade to the Fortran code listed in Fig. 4.13 results in the code listed in Fig. 4.17. Note that for each variable assignment, Tapenade added an assignment (with a `d` added to the variable name) that corresponds to the symbolic differentiation of the operators in the original assignment. The variable `xd` is the *seed vector* that determines the input variable of interest with respect to which we differentiate. In our case we want to compute all four terms in the Jacobian (4.56), so we need to run this code twice: one with `x(1)=1`, `x(2)=0`, and another with `x(1)=0`, `x(2)=1` to get the first and the second columns of the Jacobian, respectively. If desired, directional derivatives can also be computed by combining multiple nonzero entries in the seed vector.

```
FUNCTION F_D(x, xd, f)
  REAL :: x(2), xd(2)
  REAL :: det, detd
  REAL :: y(2), yd(2)
  REAL :: f(2), f_d(2)
  detd = xd(1)*x(2)**2 + x(1)*2*x(2)*xd(2)
  det = 2 + x(1)*x(2)**2
  yd = 0.0
  yd(1) = ((2*x(2)*xd(2)*SIN(x(1))+x(2)**2*xd(1)*COS(x(1)))*det-x(2)**2*&
&    SIN(x(1))*detd)/det**2
  y(1) = x(2)**2*SIN(x(1))/det
  yd(2) = (xd(1)*COS(x(1))*det-SIN(x(1))*detd)/det**2
  y(2) = SIN(x(1))/det
  f_d = 0.0
  f_d(1) = yd(1)
  f(1) = y(1)
  f_d(2) = yd(2)*SIN(x(1)) + y(2)*xd(1)*COS(x(1))
  f(2) = y(2)*SIN(x(1))
  RETURN
END FUNCTION F_D
```

Figure 4.17: Fortran code differentiated using the forward mode of the algorithmic differentiation tool Tapenade

Applying the reverse mode of Tapenade to our original Fortran code results in the code listed in Fig. 4.18. The resulting quote is more complex than the one generated in the forward mode due to the fact that the reverse mode needs to do the computations in reverse (which corresponds to the back substitution step in solving Eq. (4.58)). To complicate things further, this reverse sequence of computations requires the values of the various variables in the original code. The resulting procedure is one in which the original calculations are first performed (lines up to the assignment of `y(2)`), and only then does the reverse computations take place, ending with the final assignment of the derivatives of interest, `xb`.

In this example, no variables are overwritten. If that were the case, then Tapenade would store any intermediate values of the variables during the original computation using its own memory space in order to be able to recall these variables at the reverse computation stage.

```
SUBROUTINE F_B(x, xb, fb)
  REAL :: x(2), xb(2),
  REAL :: y(2), yb(2)
  REAL :: f(2), fb(2)
  REAL :: det, detb, tempb, temp
  det = 2 + x(1)*x(2)**2
  y(1) = x(2)**2*SIN(x(1))/det
  y(2) = SIN(x(1))/det
  xb = 0.0
  yb = 0.0
  yb(2) = yb(2) + SIN(x(1))*fb(2)
  xb(1) = xb(1) + y(2)*COS(x(1))*fb(2)
  fb(2) = 0.0
  yb(1) = yb(1) + fb(1)
  xb(1) = xb(1) + COS(x(1))*yb(2)/det
  detb = -(SIN(x(1))*yb(2)/det**2)
  yb(2) = 0.0
  tempb = SIN(x(1))*yb(1)/det
  temp = x(2)**2/det
  xb(2) = xb(2) + 2*x(2)*tempb
  detb = detb - temp*tempb
  xb(1) = xb(1) + x(2)**2*detb + temp*COS(x(1))*yb(1)
  xb(2) = xb(2) + x(1)*2*x(2)*detb
END SUBROUTINE F_B
```

Figure 4.18: Fortran code differentiated using the reverse mode of the algorithmic differentiation tool Tapenade

4.8 Analytic Methods

Analytic methods are the most accurate and efficient methods available for computing derivatives. However, analytic methods are much more involved that the other methods, since they require detailed knowledge of the computational model and a long implementation time. Analytic methods are applicable when we have a quantity of interest $\boldsymbol{f}$ that depends implicitly on the independent variables of interest $\boldsymbol{x}$, as previously described in Eq. (4.30), which we repeat here for convenience:

$$\boldsymbol{f} = \boldsymbol{F}(\boldsymbol{x}, \boldsymbol{y}(\boldsymbol{x})). \tag{4.62}$$

The implicit relationship between the state variables $\boldsymbol{y}$ and the independent variables is defined by the solution of a set of residual equations, which we also repeat here:

$$\boldsymbol{r} = \boldsymbol{R}(\boldsymbol{x}, \boldsymbol{y}(\boldsymbol{x})) = 0. \tag{4.63}$$

A graphical representation of the system of governing equations was shown previously in Fig. 4.11.

By writing the computational model in this form, we have assumed a *discrete* analytic approach. This is in contrast to the *continuous* approach, in which the equations are not discretized until later. We will not discuss the continuous approach here, but ample literature can be found on the subject [2, 18, 29, 30], including discussions comparing the two approaches [15, 49].

In this section we derive the two forms of the analytic method — the direct and the adjoint — in two ways. The first derivation follows the derivation that is typically presented in the literature, while the second derivation is based on the chain rule identity (4.40), and is a new perspective that connects it to algorithmic differentiation.

4.8.1 Traditional Derivation

As a first step toward obtaining the derivatives that we ultimately want to compute, we use the chain rule to write the total derivative Jacobian of $\boldsymbol{f}$ as

$$\frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} = \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}} + \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}} \frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}\boldsymbol{x}}, \tag{4.64}$$

where the result is an $n_f \times n_x$ matrix. As previously mentioned it important to distinguish the total and partial derivatives and define the context. The partial derivatives represent the variation of $\boldsymbol{f} = \boldsymbol{F}(\boldsymbol{x})$ with respect to changes in $\boldsymbol{x}$ for a fixed $\boldsymbol{y}$, while the total derivative $\mathrm{d}\boldsymbol{f}/\mathrm{d}\boldsymbol{x}$ takes into account the change in $\boldsymbol{y}$ that is required to keep the residual equations (4.63) equal to zero. As we have seen, this distinction depends on the context, i.e., what is considered a total or partial derivative depends on the level that is being considered in the nested system of components.

We should also mention that the partial derivatives can be computed using the methods that we have described earlier (finite differences and complex step), as well as the method that we describe in the next section (algorithmic differentiation).

Since the governing equations must always be satisfied, the total derivative of the residuals (4.63) with respect to the design variables must also be zero. Thus, using the chain rule we obtain,

$$\frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}\boldsymbol{x}} = \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}} + \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{y}} \frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}\boldsymbol{x}} = 0. \tag{4.65}$$

The computation of the total derivative matrix $\mathrm{d}\boldsymbol{y}/\mathrm{d}\boldsymbol{x}$ in Eqs. (4.64) and (4.65) has a much higher computational cost than any of the partial derivatives, since it requires the solution of the residual equations. The partial derivatives can be computed by differentiating the function $\boldsymbol{F}$ with respect to $\boldsymbol{x}$ while keeping $\boldsymbol{y}$ constant.

The linearized residual equations (4.65) provide the means for computing the total Jacobian matrix $\mathrm{d}\boldsymbol{y}/\mathrm{d}\boldsymbol{x}$, by rewriting those equations as

$$\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{y}} \frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}\boldsymbol{x}} = -\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}}. \tag{4.66}$$

Substituting this result into the total derivative Eq. (4.64), we obtain

$$\frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} = \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}} - \underbrace{\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}} \overbrace{\left[\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{y}}\right]^{-1}}^{-\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}\boldsymbol{x}}}}_{\boldsymbol{\psi}} \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}}. \tag{4.67}$$

The inverse of the square Jacobian matrix $\partial \boldsymbol{R}/\partial \boldsymbol{y}$ is not necessarily explicitly calculated. However, we use the inverse to denote the fact that this matrix needs to be solved as a linear system with some right-hand-side vector.

Eq. (4.67) shows that there are two ways of obtaining the total derivative matrix $\mathrm{d}\boldsymbol{y}/\mathrm{d}\boldsymbol{x}$, depending on which right-hand side is chosen for the solution of the linear system. Fig. 4.19 shows the sizes of the matrices in Eq. (4.67), which depend on the shape of $\mathrm{d}\boldsymbol{f}/\mathrm{d}\boldsymbol{x}$. The diagrams in Fig. 4.19 illustrate why the direct method is preferable when $n_f > n_x$ and the adjoint method is more efficient when $n_x > n_f$. This is the same fundamental difference that we observed between the forward and reverse forms of the unifying chain rule.

In this discussion, we have assumed that the governing equations have been discretized. The same kind of procedure can be applied to continuous governing equations. The principle is the

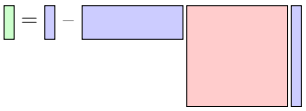
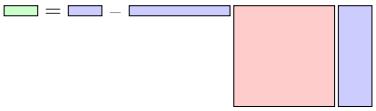
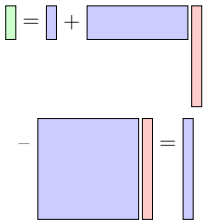
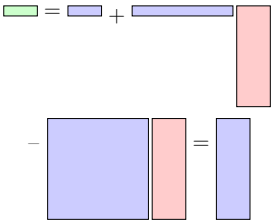
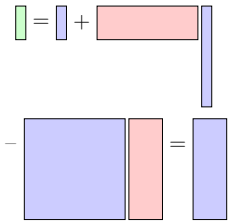
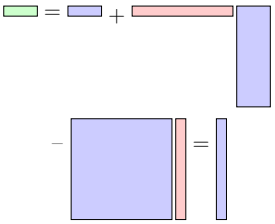
	$n_f > n_x$	$n_x > n_f$
$\frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} = \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}} - \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}} \left[\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{y}}\right]^{-1} \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}}$		
Direct method	$n_f > n_x$	$n_x > n_f$
$\frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} = \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}} + \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}} \frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}\boldsymbol{x}}$ $-\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{y}} \frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}\boldsymbol{x}} = \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}}$		
Adjoint method	$n_f > n_x$	$n_x > n_f$
$\frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} = \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}} + \frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{r}} \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}}$ $-\left[\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{y}}\right]^T \left[\frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{r}}\right]^T = \left[\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}}\right]^T$		

Figure 4.19: Block matrix diagrams illustrating the direct and adjoint methods. The matrices in blue contain partial derivatives and are relatively cheap to compute. The matrices in red contain total derivatives that are computed by solving linear systems (the third and fifth rows). In both cases, it is assumed that $n_y \gg n_x, n_f$.

same, but the notation would have to be more general. The equations, in the end, have to be discretized in order to be solved numerically. Fig. 4.20 shows the two ways of arriving at the discrete sensitivity equations. We can either differentiate the continuous governing equations first and then discretize them, or discretize the governing equations and differentiate them in the second step. The resulting sensitivity equations should be equivalent, but are not necessarily the same.

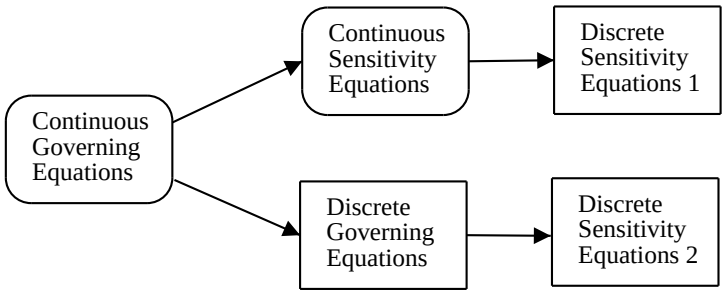


Figure 4.20: The two ways of obtaining the discretized sensitivity equations

Differentiating the continuous governing equations first is usually more involved. In addition, applying boundary conditions to the differentiated equations can be non-intuitive as some of these boundary conditions are non-physical.

Example 4.20. Direct and Adjoint Methods Applied to Finite-Element Structural Analysis

The discretized governing equations for a finite-element structural model are,

$$\mathcal{R}_k = K_{ki} u_i - F_k = 0, \tag{4.68}$$

where K_{ki} is the stiffness matrix, u_i is the vector of displacement (the state) and F_k is the vector of applied force (not to be confused with the function of interest from the previous section!).

We are interested in finding the sensitivities of the stress, which is related to the displacements by the equation,

$$\sigma_m = S_{mi} u_i. \tag{4.69}$$

We will consider the design variables to be the cross-sectional areas of the elements, A_j. We will now look at the terms that we need to use the generalized total derivative Eq. (4.67).

For the matrix of sensitivities of the governing equations with respect to the state variables we find that it is simply the stiffness matrix, i.e.,

$$\frac{\partial \mathcal{R}_k}{\partial y_i} = \frac{\partial (K_{ki} u_i - F_k)}{\partial u_i} = K_{ki}. \tag{4.70}$$

Lets consider the derivative of the residuals with respect to the design variables (cross-sectional areas in our case). Neither the displacements of the applied forces vary explicitly with the element sizes. The only term that depends on A_j directly is the stiffness matrix, so we get,

$$\frac{\partial \mathcal{R}_k}{\partial x_j} = \frac{\partial (K_{ki} u_i - F_k)}{\partial A_j} = \frac{\partial K_{ki}}{\partial A_j} u_i \tag{4.71}$$

The partial derivative of the stress with respect to the displacements is simply given by the matrix in Eq. (4.69), i.e.,

$$\frac{\partial f_m}{\partial y_i} = \frac{\partial \sigma_m}{\partial u_i} = S_{mi} \tag{4.72}$$

Finally, the explicit variation of stress with respect to the cross-sectional areas is zero, since the stresses depends only on the displacement field,

$$\frac{\partial f_m}{\partial x_j} = \frac{\partial \sigma_m}{\partial A_j} = 0. \tag{4.73}$$

Substituting these into the generalized total derivative (Eq. (4.67)) we get:

$$\frac{\mathrm{d}\sigma_m}{\mathrm{d}A_j} = -S_{mi} K_{ki}^{-1} \frac{\partial K_{ki}}{\partial A_j} u_i \tag{4.74}$$

Referring to the theory presented previously, if we were to use the direct method, we would solve,

$$K_{ki} \frac{\mathrm{d}u_i}{\mathrm{d}A_j} = -\frac{\partial K_{ki}}{\partial A_j} u_i \tag{4.75}$$

and then substitute the result in,

$$\frac{\mathrm{d}\sigma_m}{\mathrm{d}A_j} = S_{mi} \frac{\mathrm{d}u_i}{\mathrm{d}A_j} \tag{4.76}$$

to calculate the desired derivatives.

The adjoint method could also be used, in which case we would solve Eq. (4.89) for the structures case,

$$K_{ki}^T \psi_k = -S_{mi} \tag{4.77}$$

Then we would substitute the adjoint vector into the equation,

$$\frac{\mathrm{d}\sigma_m}{\mathrm{d}A_j} = \psi_k^T \frac{\partial K_{ki}}{\partial A_j} u_i. \tag{4.78}$$

to calculate the desired derivatives.

4.8.2 Derivation of the Direct and Adjoint Methods from the Unifying Chain Rule

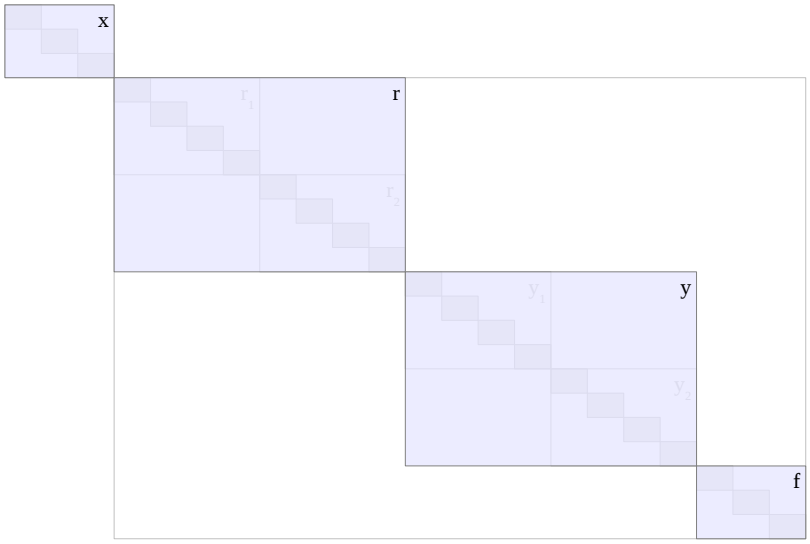
The first step in this derivation is to clearly define the level of decomposition, i.e., how the variables $\boldsymbol{v}$ are defined. Since the analytic methods apply to coupled systems of equations, the assumption that the Jacobians are lower triangular matrices does no longer apply. Therefore, we first linearize the residuals (4.63) so that it is possible to write explicit equations for the state variables $\boldsymbol{y}$. We linearize about the converged point $[\boldsymbol{x}_0, \boldsymbol{r}_0, \boldsymbol{y}_0, \boldsymbol{f}_0]^T$, and divide $\boldsymbol{v}$ into

$$\boldsymbol{v}_1 = \boldsymbol{x}, \quad \boldsymbol{v}_2 = \boldsymbol{r}, \quad \boldsymbol{v}_3 = \boldsymbol{y}, \quad \boldsymbol{v}_4 = \boldsymbol{f}. \tag{4.79}$$

So instead of defining them as every single variable assignment in the computer program, we defined them as variations in the design variables, residuals, state variables and quantities of interest. This decomposition is represented in Fig. 4.21. The dependence of these variations about the converged states is illustrated in Fig. 4.22.

Since $\boldsymbol{x}$ are the only independent variables, we have an initial perturbation $\boldsymbol{x}$ that leads to a response $\boldsymbol{r}$. However, we require that $\boldsymbol{r} = 0$ be satisfied when we take a total derivative, and therefore,

$$\boldsymbol{R} = 0 \quad \Rightarrow \quad \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}} \boldsymbol{x} + \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{y}} \boldsymbol{y} = 0 \tag{4.80}$$



$$\boldsymbol{v} = [\underbrace{v_1, \ldots, v_{n_x}}_{\boldsymbol{x}}, \underbrace{v_{(n_x+1)}, \ldots, v_{(n_x+n_y)}}_{\boldsymbol{r}}, \underbrace{v_{(n_x+n_y+1)}, \ldots, v_{(n_x+2n_y)}}_{\boldsymbol{y}}, \underbrace{v_{(n-n_f)}, \ldots, t_n}_{\boldsymbol{f}}]^T.$$

Figure 4.21: Decomposition level for analytic methods

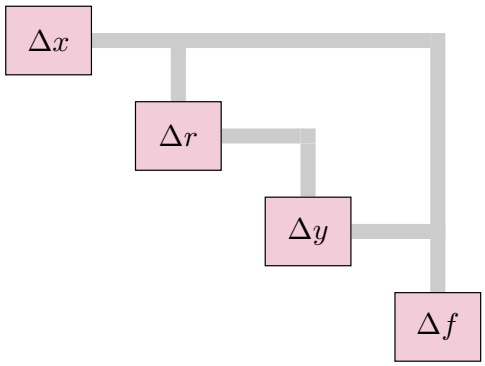


Figure 4.22: Dependence of the variations in the design variables, residuals, states and quantities of interest for the linearized system

The solution vector $\boldsymbol{y}$ from this linear system is used in conjunction with the original perturbation

vector $\boldsymbol{x}$ to compute the total change in $\boldsymbol{f}$, i.e.,

$$\boldsymbol{v}_1 = \boldsymbol{x} \tag{4.81}$$

$$\boldsymbol{v}_2 = \boldsymbol{r} = \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}} \boldsymbol{x} \tag{4.82}$$

$$\boldsymbol{v}_3 = \boldsymbol{y} = \left[\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{y}}\right]^{-1} (-\boldsymbol{r}) \tag{4.83}$$

$$\boldsymbol{v}_4 = \boldsymbol{f} = \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}} \boldsymbol{x} + \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}} \boldsymbol{y} \tag{4.84}$$

$$\tag{4.85}$$

At this point, all variables are functions of only previous variables, so we can apply the forward and reverse chain rule equations (4.40) to the linearized system with the definition (4.79). The result is the set of block matrix equations shown in Fig. 4.23. The forward chain rule (4.38) yields the left column, which is the direct method. The right column represents the adjoint method, which is obtained from the reverse chain rule (4.39).

4.8.3 Direct Method

The direct method involves solving the linear system with $-\partial \boldsymbol{R}/\partial \boldsymbol{x}$ as the right-hand side vector, which results in the linear system (4.66). This linear system needs to be solved for n_x right-hand sides to get the full Jacobian matrix $\mathrm{d}\boldsymbol{y}/\,\mathrm{d}\boldsymbol{x}$. Then, we can use $\mathrm{d}\boldsymbol{y}/\,\mathrm{d}\boldsymbol{x}$ in Eq. (4.64) to obtain the derivatives of interest, $\mathrm{d}\boldsymbol{f}/\,\mathrm{d}\boldsymbol{x}$.

Since the cost of computing derivatives with the direct method is proportional to the number of design variables, n_x, it does not have much of a computational cost advantage relative to finite differencing. In a case where the computational model is a nonlinear system, the direct method can be advantageous. Both methods require the solution of a system with the same size n_x times, but the direct method solves a linear system, while the finite-difference method solves the original nonlinear one.

Example 4.21. Direct Method Numerical Example

We now return to the numerical example introduced in Section 13 and apply the direct method. Since there are just two state variables, Eq. (4.66) is the following 2×2 linear system:

$$\begin{bmatrix} -\dfrac{\partial R_1}{\partial y_1} & -\dfrac{\partial R_1}{\partial y_2} \\ -\dfrac{\partial R_2}{\partial y_1} & -\dfrac{\partial R_2}{\partial y_2} \end{bmatrix} \begin{bmatrix} \dfrac{\mathrm{d}y_1}{\mathrm{d}x_1} & \dfrac{\mathrm{d}y_1}{\mathrm{d}x_2} \\ \dfrac{\mathrm{d}y_2}{\mathrm{d}x_1} & \dfrac{\mathrm{d}y_2}{\mathrm{d}x_2} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial R_1}{\partial x_1} & \dfrac{\partial R_1}{\partial x_2} \\ \dfrac{\partial R_2}{\partial x_1} & \dfrac{\partial R_2}{\partial x_2} \end{bmatrix}.$$

In a more realistic example, computation of the partial derivatives would be non-trivial as the residuals typically do not have simple analytical expressions. In this case, the residuals do have simple forms, so we can use symbolic differentiation to compute each partial derivative to obtain

$$\begin{bmatrix} -x_1 & -2 \\ 1 & -x_2^2 \end{bmatrix} \begin{bmatrix} \dfrac{\mathrm{d}y_1}{\mathrm{d}x_1} & \dfrac{\mathrm{d}y_1}{\mathrm{d}x_2} \\ \dfrac{\mathrm{d}y_2}{\mathrm{d}x_1} & \dfrac{\mathrm{d}y_2}{\mathrm{d}x_2} \end{bmatrix} = \begin{bmatrix} y_1 - \cos x_1 & 0 \\ 0 & 2x_2 y_2 \end{bmatrix}.$$

Though we are not interesting in finding any derivatives with respect to x_2, both right hand sides of the system are shown for completeness. Since the analytic methods are derived based on a

$$\begin{bmatrix} \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ -\dfrac{\partial \boldsymbol{V}_2}{\partial \boldsymbol{v}_1} & \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} \\ -\dfrac{\partial \boldsymbol{V}_3}{\partial \boldsymbol{v}_1} & -\dfrac{\partial \boldsymbol{V}_3}{\partial \boldsymbol{v}_2} & \boldsymbol{I} & \boldsymbol{0} \\ -\dfrac{\partial \boldsymbol{V}_4}{\partial \boldsymbol{v}_1} & -\dfrac{\partial \boldsymbol{V}_4}{\partial \boldsymbol{v}_2} & -\dfrac{\partial \boldsymbol{V}_4}{\partial \boldsymbol{v}_3} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{I} \\ \dfrac{\mathrm{d}\boldsymbol{v}_2}{\mathrm{d}\boldsymbol{v}_1} \\ \dfrac{\mathrm{d}\boldsymbol{v}_3}{\mathrm{d}\boldsymbol{v}_1} \\ \dfrac{\mathrm{d}\boldsymbol{v}_4}{\mathrm{d}\boldsymbol{v}_1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{I} \\ \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{0} \end{bmatrix}$$

(a) Forward chain rule

$$\begin{bmatrix} \boldsymbol{I} & -\left[\dfrac{\partial \boldsymbol{V}_2}{\partial \boldsymbol{v}_1}\right]^T & -\left[\dfrac{\partial \boldsymbol{V}_3}{\partial \boldsymbol{v}_1}\right]^T & -\left[\dfrac{\partial \boldsymbol{V}_4}{\partial \boldsymbol{v}_1}\right]^T \\ \boldsymbol{0} & \boldsymbol{I} & -\left[\dfrac{\partial \boldsymbol{V}_3}{\partial \boldsymbol{v}_2}\right]^T & -\left[\dfrac{\partial \boldsymbol{V}_4}{\partial \boldsymbol{v}_2}\right]^T \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} & -\left[\dfrac{\partial \boldsymbol{V}_4}{\partial \boldsymbol{v}_3}\right]^T \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \dfrac{\mathrm{d}\boldsymbol{v}_4}{\mathrm{d}\boldsymbol{v}_1} \\ \dfrac{\mathrm{d}\boldsymbol{v}_4}{\mathrm{d}\boldsymbol{v}_2} \\ \dfrac{\mathrm{d}\boldsymbol{v}_4}{\mathrm{d}\boldsymbol{v}_3} \\ \boldsymbol{I} \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{I} \end{bmatrix}$$

(b) Reverse chain rule

$$\begin{bmatrix} \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ -\dfrac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}} & \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \left[\dfrac{\partial \boldsymbol{R}}{\partial \boldsymbol{y}}\right]^{-1} & \boldsymbol{I} & \boldsymbol{0} \\ -\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}} & \boldsymbol{0} & -\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{I} \\ \dfrac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}\boldsymbol{x}} \\ \dfrac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}\boldsymbol{x}} \\ \dfrac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{I} \\ \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{0} \end{bmatrix}$$

(c) Forward chain rule (simplified)

$$\begin{bmatrix} \boldsymbol{I} & -\left[\dfrac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}}\right]^T & \boldsymbol{0} & -\left[\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right]^T \\ \boldsymbol{0} & \boldsymbol{I} & \left[\dfrac{\partial \boldsymbol{R}}{\partial \boldsymbol{y}}\right]^{-T} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} & -\left[\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}}\right]^T \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \dfrac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} \\ \dfrac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{r}} \\ \dfrac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{y}} \\ \boldsymbol{I} \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{I} \end{bmatrix}$$

(d) Reverse chain rule (simplified)

$$\begin{aligned} \frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}\boldsymbol{x}} &= \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}} \\ \frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}\boldsymbol{x}} &= -\left[\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{y}}\right]^{-1} \frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}\boldsymbol{x}} \\ \frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} &= \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}} + \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}} \frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}\boldsymbol{x}} \end{aligned}$$

(e) Forward substituted (direct method)

$$\begin{aligned} \frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{y}} &= \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}} \\ \frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{r}} &= -\frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{y}} \left[\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{y}}\right]^{-1} \\ \frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} &= \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}} + \frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{r}} \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}} \end{aligned}$$

(f) Back substituted (adjoint method)

$$\begin{bmatrix} \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} \\ -\dfrac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}} & -\dfrac{\partial \boldsymbol{R}}{\partial \boldsymbol{y}} & \boldsymbol{0} \\ -\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}} & -\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{I} \\ \dfrac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}\boldsymbol{x}} \\ \dfrac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{I} \\ \boldsymbol{0} \\ \boldsymbol{0} \end{bmatrix}$$

(g) Direct method with the inverse Jacobian eliminated

$$\begin{bmatrix} \boldsymbol{I} & -\left[\dfrac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}}\right]^T & -\left[\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right]^T \\ \boldsymbol{0} & -\left[\dfrac{\partial \boldsymbol{R}}{\partial \boldsymbol{y}}\right]^T & -\left[\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}}\right]^T \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \dfrac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} \\ \dfrac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{r}} \\ \boldsymbol{I} \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{I} \end{bmatrix}$$

(h) Adjoint method with the inverse Jacobian eliminated

Figure 4.23: Derivation of the direct (left) and adjoint (right) methods from the forward and reverse chain rule, respectively. The top row shows the chain rule in block form with the four variable vectors; in the second row we replace those vectors with the variables we defined, and the third row shows the equations after the solution of the block matrix, which correspond to the traditional direct and adjoint equations. In the last row, the direct and adjoint methods are presented with the inverse of the $\partial \boldsymbol{R}/\partial \boldsymbol{y}$ matrix eliminated, at the cost of the overall matrix no longer being lower/upper triangular.

linearization of the system at a converged state, we must evaluate the system at $[x_1, x_2] = [1, 1]$ and $[y_1, y_2] = [\frac{\sin 1}{3}, \frac{\sin 1}{3}]$. The computed values for dy_1/dx_1 and dy_2/dx_1 can be used to find df_1/dx_1 using the following equation:

$$\frac{\mathrm{d}f_1}{\mathrm{d}x_1} = \frac{\partial F_1}{\partial x_1} + \frac{\partial F_1}{\partial y_1}\frac{\mathrm{d}y_1}{\mathrm{d}x_1} + \frac{\partial F_1}{\partial y_2}\frac{\mathrm{d}y_2}{\mathrm{d}x_1}.$$

4.8.4 Adjoint Method

Returning to the total derivative Eq. (4.67), we observe that there is an alternative option for computing the total derivatives; the linear system involving the square Jacobian matrix $\partial \boldsymbol{R}/\partial \boldsymbol{y}$ can be solved with $\partial \boldsymbol{f}/\partial \boldsymbol{y}$ as the right-hand side. This results in the following linear system, which we call the *adjoint equations*,

$$\left[\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{y}}\right]^T \boldsymbol{\psi} = -\left[\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}}\right]^T, \tag{4.86}$$

where we will call $\boldsymbol{\psi}$ the *adjoint matrix* (of size $n_y \times n_f$). Although this is usually expressed as a vector, we obtain a matrix due to our generalization for the case where $\boldsymbol{f}$ is a vector. The solution of this linear system needs to be solved for each column of $[\partial \boldsymbol{F}/\partial \boldsymbol{y}]^T$, and thus the computational cost is proportional to the number of quantities of interest, n_f. The adjoint vector can then be substituted into Eq. (4.67) to find the total derivative,

$$\frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} = \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}} + \boldsymbol{\psi}^T \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}} \tag{4.87}$$

Thus, the cost of computing the total derivative matrix using the adjoint method is independent of the number of design variables, n_x, and instead proportional to the number of quantities of interest, $\boldsymbol{f}$.

Note that the partial derivatives shown in these equations need to be computed using some other method. They can be differentiated symbolically, computed by finite differences, the complex-step method or even AD. The use of AD for these partials has been shown to be particularly effective in the development of analytic methods for PDE solvers [41].

Example 4.22. Adjoint Method Numerical Example

We now apply the adjoint method to compute df_1/dx_1 in the numerical example from Section 13. In this case, the linear system is

$$\begin{bmatrix} -\dfrac{\partial R_1}{\partial y_1} & -\dfrac{\partial R_2}{\partial y_1} \\ -\dfrac{\partial R_1}{\partial y_2} & -\dfrac{\partial R_2}{\partial y_2} \end{bmatrix} \begin{bmatrix} \dfrac{\mathrm{d}f_1}{\mathrm{d}r_1} & \dfrac{\mathrm{d}f_2}{\mathrm{d}r_1} \\ \dfrac{\mathrm{d}f_1}{\mathrm{d}r_2} & \dfrac{\mathrm{d}f_2}{\mathrm{d}r_2} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial F_1}{\partial y_1} & \dfrac{\partial F_2}{\partial y_1} \\ \dfrac{\partial F_1}{\partial y_2} & \dfrac{\partial F_2}{\partial y_2} \end{bmatrix}$$

With the partial derivatives computed symbolically the linear system becomes

$$\begin{bmatrix} -x_1 & 1 \\ -2 & -x_2^2 \end{bmatrix} \begin{bmatrix} \dfrac{\mathrm{d}f_1}{\mathrm{d}r_1} & \dfrac{\mathrm{d}f_2}{\mathrm{d}r_1} \\ \dfrac{\mathrm{d}f_1}{\mathrm{d}r_2} & \dfrac{\mathrm{d}f_2}{\mathrm{d}r_2} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & \sin x_1 \end{bmatrix}$$

After evaluating the system at $[x_1, x_2] = [1, 1]$ and $[y_1, y_2] = [\frac{\sin 1}{3}, \frac{\sin 1}{3}]$, we can find df_1/dx_1 using the computed values for df_1/dr_1 and df_1/dr_2:

$$\frac{\mathrm{d}f_1}{\mathrm{d}x_1} = \frac{\partial F_1}{\partial x_1} + \frac{\mathrm{d}f_1}{\mathrm{d}r_1}\frac{\partial R_1}{\partial x_1} + \frac{\mathrm{d}f_1}{\mathrm{d}r_2}\frac{\partial R_2}{\partial x_1}$$

4.8.5 Direct vs. Adjoint

We can now see that the choice of the solution procedure (direct vs. adjoint) to obtain the total sensitivity (4.67) has a substantial impact on the cost of sensitivity analysis. Although all the partial derivative terms are the same for both the direct and adjoint methods, the order of the operations is not. Notice that once $\mathrm{d}y_i/\,\mathrm{d}x_n$ is computed, it is valid for any function f, but must be recomputed for each design variable (direct method). On the other hand, Ψ_k is valid for all design variables, but must be recomputed for each function (adjoint method).

The cost involved in calculating sensitivities using the adjoint method is therefore practically independent of the number of design variables. After having solved the governing equations, the adjoint equations are solved only once for each f. Moreover, the cost of solution of the adjoint equations is similar to that of the solution of the governing equations since they are of similar complexity and the partial derivative terms are easily computed.

Therefore, if the number of design variables is greater than the number of functions for which we seek sensitivity information, the adjoint method is computationally more efficient. Otherwise, if the number of functions to be differentiated is greater than the number of design variables, the direct method would be a better choice.

A comparison of the cost of computing sensitivities with the direct versus adjoint methods is shown in Table 4.5. With either method, we must factorize the same matrix, $\partial\mathcal{R}_k/\partial y_i$. The difference in the cost comes form the back-solve step for solving equations (4.66) and (4.89) respectively. The direct method requires that we perform this step for each design variable (i.e. for each j) while the adjoint method requires this to be done for each function of interest (i.e. for each i). The multiplication step is simply the calculation of the final sensitivity expressed in equations (4.66) and (4.89) respectively. The cost involved in this step when computing the same set of sensitivities is the same for both methods.

Table 4.5: Cost comparison of computing sensitivities for direct and adjoint methods

Step	Direct	Adjoint
Factorization	same	same
Back-solve	N_x times	N_f times
Multiplication	same	same

Example 4.23. The Automatic Differentiation Adjoint (ADjoint) Method [42]

The automatic differentiation adjoint (ADjoint) method is a hybrid between automatic differentiation and the adjoint method. In a nutshell, we take the adjoint equations formulated above, compute the partial derivatives in those equations with automatic differentiation, and then solve the linear system and perform the necessary matrix-vector products.

We chose to use Tapenade as it is the only non-commercial tool with support for Fortran 90. Tapenade is the successor of Odyssée [16] and was developed at the INRIA. It uses source transformation and can perform differentiation in either forward or reverse mode. Furthermore, the Tapenade team is actively developing their software and has been very responsive to a number of suggestions towards completing their support of the Fortran 90 standard.

In order to verify the results given by the ADjoint approach we decided to use the complex-step derivative approximation [38, 46] as our benchmark.

In semi-discrete form the Euler equations are,

$$\frac{\mathrm{d}w_{ijk}}{\mathrm{d}t} + \mathcal{R}_{ijk}(w) = 0, \tag{4.88}$$

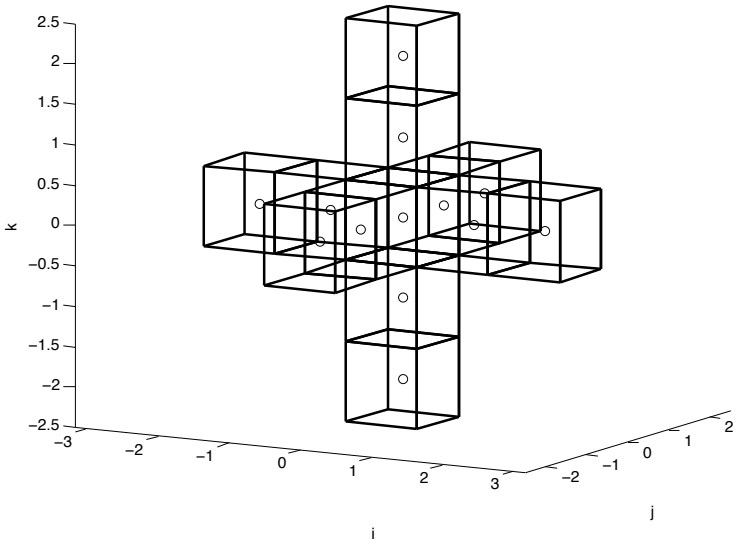


Figure 4.24: Stencil for the residual computation

where $\mathcal{R}$ is the residual described earlier with all of its components (fluxes, boundary conditions, artificial dissipation, etc.).

The adjoint equations (4.86) can be re-written for this flow solver as,

$$\left[\frac{\partial \mathcal{R}}{\partial w}\right]^T \psi = -\frac{\partial I}{\partial w}. \tag{4.89}$$

where ψ is the *adjoint vector*. The total derivative (4.67) in this case is,

$$\frac{\mathrm{d}I}{\mathrm{d}x} = \frac{\partial I}{\partial x} + \psi^T \frac{\partial \mathcal{R}}{\partial x}. \tag{4.90}$$

We propose to compute the partial derivative matrices $\partial\mathcal{R}/\partial w$, $\partial I/\partial w$, $\partial I/\partial x$ and $\partial\mathcal{R}/\partial x$ using automatic differentiation instead of using manual differentiation or using finite differences. Where appropriate we will use the reverse mode of automatic differentiation.

To better understand the choices made in the mode of differentiation as well as the effect of these choices in the general case we define the following numbers:

N_c**:** The number of cells in the domain. For three-dimensional domains where the Navier–Stokes equations are solved, this can be $\mathcal{O}(10^6)$.

N_s**:** The number of cells in the stencil whose variables affect the residual of a given cell. In our case, we consider inviscid and dissipation fluxes, so the stencil is as shown in Fig. 4.24 and $N_s = 13$.

N_w**:** The number of flow variables (and also residuals) for each cell. In our case $N_w = 5$.

In principle, because $\partial R/\partial w$ is a square matrix, neither mode should have an advantage over the other in terms of computational time. However, due to the way residuals are computed, the reverse mode is much more efficient. To explain the reason for this, consider a very simplified version of a calculation resembling the residual calculation in CFD shown in Fig. 4.25. This subroutine loops through a two-dimensional domain and computes `r` (the "residual") in the interior of that domain.

```
SUBROUTINE RESIDUAL(w, r, ni, nj)
  IMPLICIT NONE
  INTEGER :: ni, nj
  REAL :: r(ni, nj), w(0:ni+1, 0:nj+1)
  INTEGER :: i, j
  INTRINSIC SQRT
  DO i=1,ni
    DO j=1,nj
      w(i, j) = w(i, j) + SQRT(w(i, j-1)*w(i, j+1))
      r(i, j) = w(i, j)*w(i-1, j) + SQRT(w(i+1, j)) + &
&               w(i, j-1)*w(i, j+1)
    END DO
  END DO
END SUBROUTINE RESIDUAL
```

Figure 4.25: Simplified subroutine for residual calculation

The residual at any cell depends only on the `w`s (the "flow variables") at that cell and at the cells immediately adjacent to it, thus the stencil of dependence forms a cross with five cells.

The residual computation in our three-dimensional CFD solver is obviously much more complicated: It involves multiple subroutines, a larger stencil, the computation of the different fluxes and applies many different types of boundary conditions. However, this simple example is sufficient to demonstrate the computational inefficiencies of a purely automatic approach.

Forward mode differentiation was used to produce the subroutine shown in Fig. 4.26. Two new variables are introduced: `wd`, which is the seed vector and `rd`, which is the gradient of all `r`s in the direction specified by the seed vector. For example, if we want the derivative with respect to `w(1,1)`, we would set `wd(1,1)=1` and all other `wd`s to zero. One can only choose one direction at a time, although Tapenade can be run in "vectorial" mode to get the whole vector of derivatives. In this case, an additional loop inside the nested loop and additional storage are required. For our purposes, the differentiated code would have to be called $N_c \times N_w$ times.

The subroutine produced by reverse mode differentiation is shown in Fig. 4.27. This case shows the additional storage requirements: Since the `w`s are overwritten, the old values must be stored for later use in the reversed loop.

The overwriting of the flow variables in the first nested loops of the original subroutine is characteristic of iterative solvers. Whenever overwriting is present, the reverse mode needs to store the time history of the intermediate variables. Tapenade provides functions (`PUSHREAL` and `POPREAL`) to do this. In this case, we can see that the `w`s are stored before they are modified in the forward sweep, and then retrieved in the reverse sweep.

On this basis, reverse mode was used to produce adjoint code for this set of routines. The call graph for the original and the differentiated routines are shown in Figs. 4.28 and 4.29, respectively.

The test case is the Lockheed-Air Force-NASA-NLR (LANN) wing [59], which is a supercritical transonic transport wing. A symmetry boundary condition is used at the root and a linear pressure extrapolation boundary condition is used on the wing surface. The freestream Mach number is 0.621. The mesh for this test case is shown in Fig. 4.30.

In Table 4.6 we show the derivatives of both drag and lift coefficients with respect to freestream Mach number for different cases.

We can see that the adjoint derivatives for these cases are extremely accurate, yielding between twelve and fourteen digits agreement when compared to the complex-step results. This is consistent with the convergence tolerance that was specified in PETSc for the adjoint solution.

To analyze the performance of the ADjoint solver, several timings were performed. They are shown in Table 4.7 for the three cases mentioned above. The coarse grid has $5,120$ flow variables,

```
SUBROUTINE RESIDUAL_D(w, wd, r, rd, ni, nj)
  IMPLICIT NONE
  INTEGER :: ni, nj
  REAL :: r(ni, nj), rd(ni, nj)
  REAL :: w(0:ni+1, 0:nj+1), wd(0:ni+1, 0:nj+1)
  INTEGER :: i, j
  REAL :: arg1, arg1d, result1, result1d
  INTRINSIC SQRT

  rd(1:ni, 1:nj) = 0.0
  DO i=1,ni
    DO j=1,nj
      arg1d = wd(i, j-1)*w(i, j+1) + w(i, j-1)*wd(i, j+1)
      arg1 = w(i, j-1)*w(i, j+1)
      IF (arg1d .EQ. 0.0 .OR. arg1 .EQ. 0.0) THEN
        result1d = 0.0
      ELSE
        result1d = arg1d/(2.0*SQRT(arg1))
      END IF
      result1 = SQRT(arg1)
      wd(i, j) = wd(i, j) + result1d
      w(i, j) = w(i, j) + result1
      IF (wd(i+1, j) .EQ. 0.0 .OR. w(i+1, j) .EQ. 0.0) THEN
        result1d = 0.0
      ELSE
        result1d = wd(i+1, j)/(2.0*SQRT(w(i+1, j)))
      END IF
      result1 = SQRT(w(i+1, j))
      rd(i, j) = wd(i, j)*w(i-1, j) + w(i, j)*wd(i-1, j) + &
&                result1d + wd(i, j-1)*w(i, j+1) + &
&                w(i, j-1)*wd(i, j+1)
      r(i, j) = w(i, j)*w(i-1, j) + result1 + &
&               w(i, j-1)*w(i, j+1)
    END DO
  END DO
END SUBROUTINE RESIDUAL_D
```

Figure 4.26: Subroutine differentiated using the forward mode

```
SUBROUTINE RESIDUAL_B(w, wb, r, rb, ni, nj)
  IMPLICIT NONE
  INTEGER :: ni, nj
  REAL :: r(ni, nj), rb(ni, nj)
  REAL :: w(0:ni+1, 0:nj+1), wb(0:ni+1, 0:nj+1)
  INTEGER :: i, j
  REAL :: tempb
  INTRINSIC SQRT
  DO i=1,ni
    DO j=1,nj
      CALL PUSHREAL4(w(i, j))
      w(i, j) = w(i, j) + SQRT(w(i, j-1)*w(i, j+1))
    END DO
  END DO
  wb(0:ni+1, 0:nj+1) = 0.0
  DO i=ni,1,-1
    DO j=nj,1,-1
      wb(i, j) = wb(i, j) + w(i-1, j)*rb(i, j)
      wb(i-1, j) = wb(i-1, j) + w(i, j)*rb(i, j)
      wb(i+1, j) = wb(i+1, j) + rb(i, j)/(2.0*SQRT(w(i+1, j)))
      wb(i, j-1) = wb(i, j-1) + w(i, j+1)*rb(i, j)
      wb(i, j+1) = wb(i, j+1) + w(i, j-1)*rb(i, j)
      rb(i, j) = 0.0
      CALL POPREAL4(w(i, j))
      tempb = wb(i, j)/(2.0*SQRT(w(i, j-1)*w(i, j+1)))
      wb(i, j-1) = wb(i, j-1) + w(i, j+1)*tempb
      wb(i, j+1) = wb(i, j+1) + w(i, j-1)*tempb
    END DO
  END DO
END SUBROUTINE RESIDUAL_B
```

Figure 4.27: Subroutine differentiated using the reverse mode

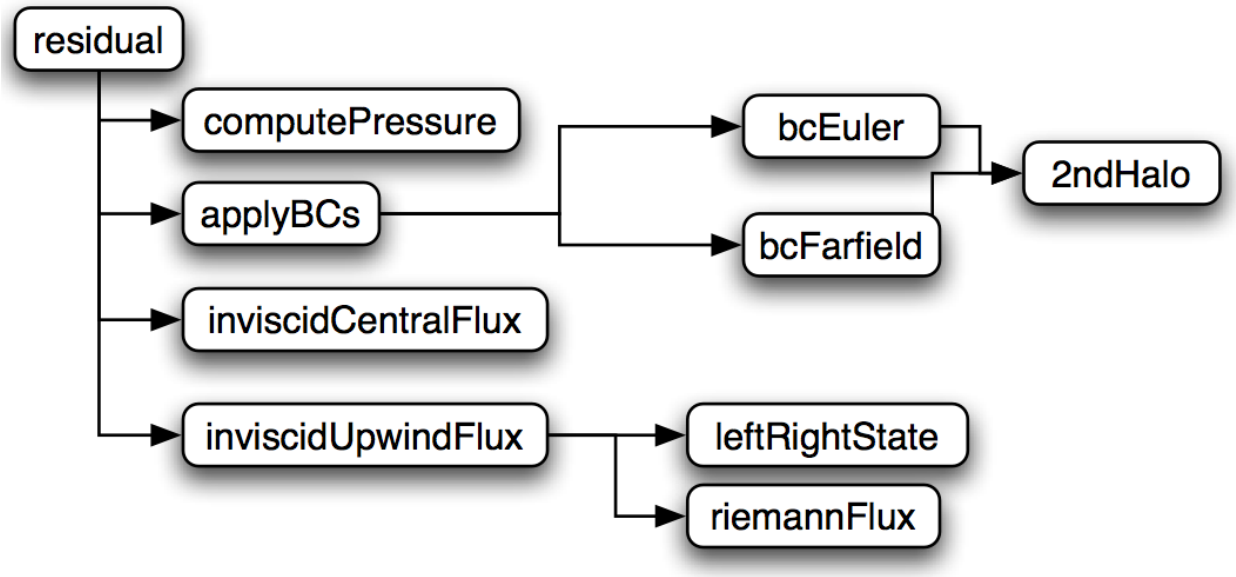


Figure 4.28: Original residual calculation

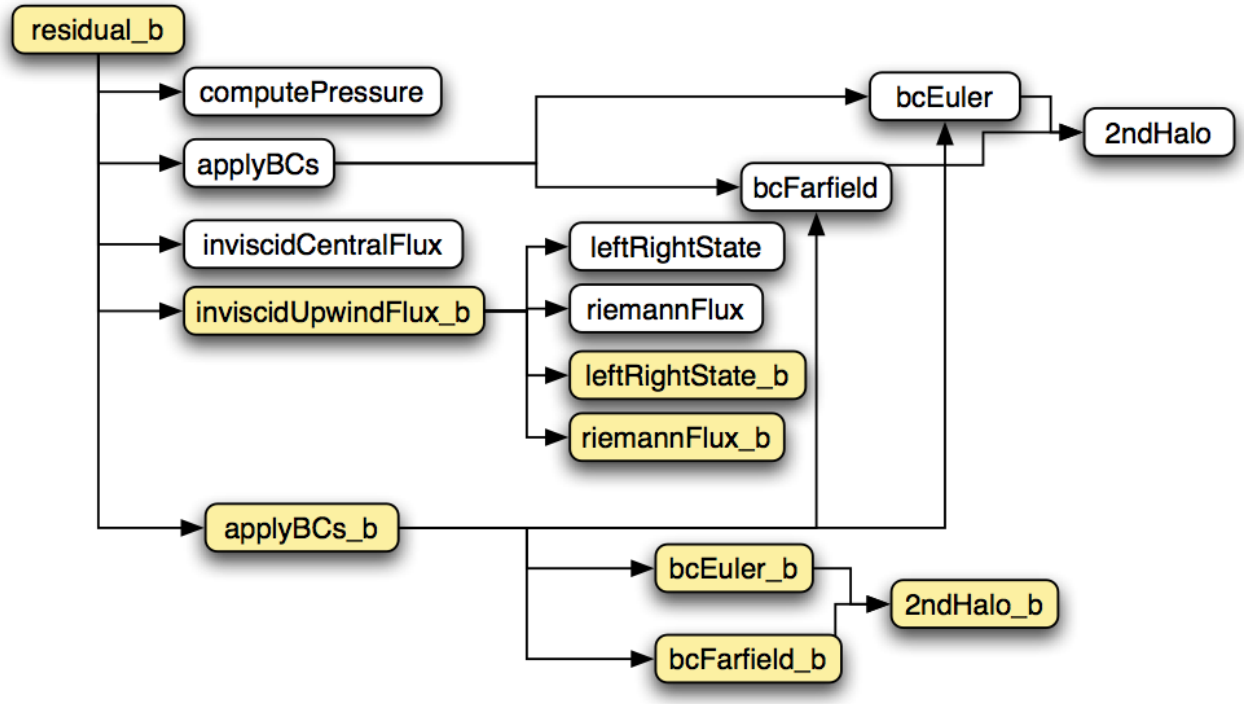


Figure 4.29: Differentiated residual calculation

the fine grid has 203,840 flow variables and the wing grid has 108,800 flow variables.

The total cost of the adjoint solver, including the computation of all the partial derivatives and the solution of the adjoint system, is less than one fourth the cost of the flow solution for the fine bump case and less that one eighth the cost of the flow solution for the wing case. This is even better than what is usually observed in the case of adjoint solvers developed by conventional means, showing that the ADjoint approach is indeed very efficient.

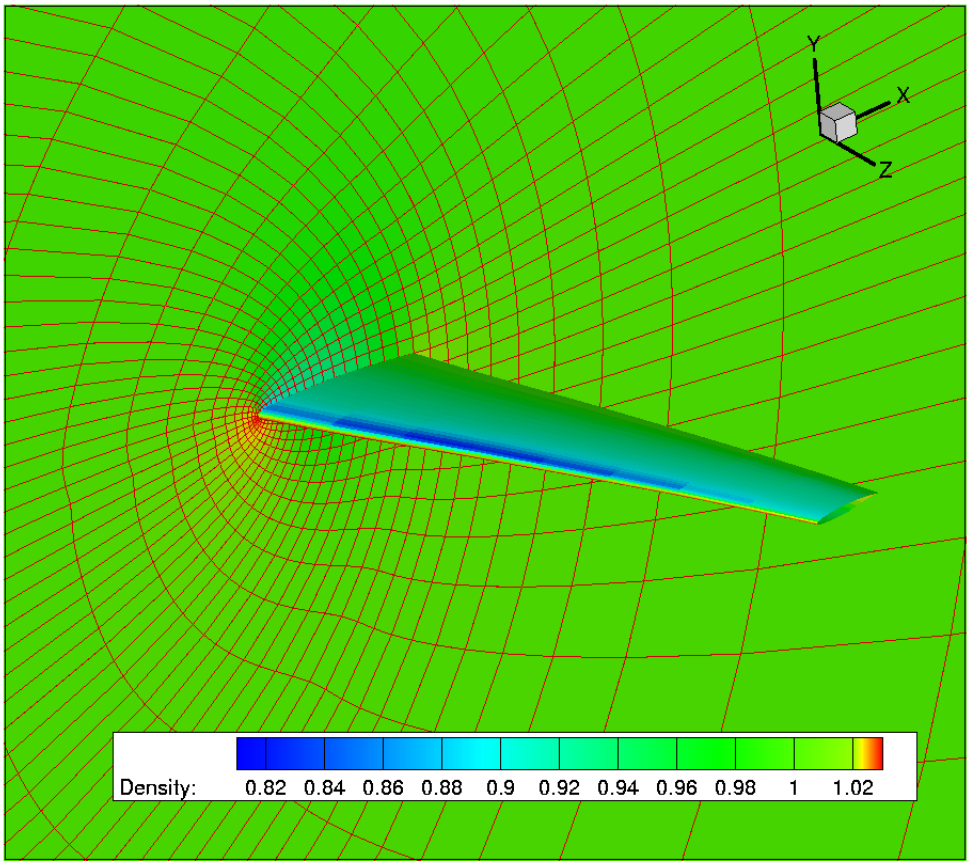


Figure 4.30: Wing mesh and density distribution

Table 4.6: Derivatives of drag and lift coefficients with respect to M_∞

Mesh	Coefficient	Inflow direction	ADjoint	Complex step
Coarse	C_D	(1,0,0)	-0.289896632731*64*	-0.289896632731*59*
	C_L		-0.267704455366*714*	-0.267704455366*683*
Fine	C_D	(1,0,0)	-0.0279501183024705	-0.0279501183024709
	C_L		0.581286047347*07*	0.581286047347*08*
Coarse	C_D	(1,0.05,0)	-0.27890764583378*6*	-0.27890764583379*2*
	C_L		-0.26208631523*3911*	-0.26208631523*3875*
Fine	C_D	(1,0.05,0)	-0.06155986310604*38*	-0.06155986310604*44*
	C_L		-0.36479675465278*7*	-0.36479675465279*7*
Wing	C_D	(1, 0.0102,0)	0.00942875710535*217*	0.00942875710535*312*
	C_L		0.267882125954*74*	0.267882125954*68*

Table 4.7: ADjoint computational cost breakdown (times in seconds)

	Fine	Wing
Flow solution	219.215	182.653
ADjoint	51.959	20.843
Breakdown:		
Setup PETSc variables	0.011	0.004
Compute flux Jacobian	11.695	5.870
Compute RHS	8.487	2.232
Solve the adjoint equations	28.756	11.213
Compute the total derivative	3.010	1.523

Bibliography

[1] Tools for automatic differentiation. URL http://www.sc.rwth-aachen.de/Research/AD/subject.html.

[2] W. K. Anderson and V. Venkatakrishnan. Aerodynamic design optimization on unstructured grids with a continuous adjoint formulation. *Computers and Fluids*, 28(4):443–480, 1999.

[3] W. K. Anderson, James C. Newman, David L. Whitfield, and Eric J. Nielsen. Sensitivity analysis for the Navier–Stokes equations on unstructured meshes using complex variables. *AIAA Paper* 99-3294, 1999.

[4] Manuel Barcelos, Henri Bavestrello, and Kurt Maute. A Schur–Newton–Krylov solver for steady-state aeroelastic analysis and design sensitivity analysis. *Computer Methods in Applied Mechanics and Engineering*, 195:2050—2069, 2006.

[5] J.-F.M. Barthelemy and J. Sobieszczanski-Sobieski. Optimum sensitivity derivatives of objective functions in nonlinear programming. *AIAA Journal*, 21:913–915, 1982.

[6] Thomas Beck. Automatic differentiation of iterative processes. *Journal of Computational and Applied Mathematics*, 50:109–118, 1994.

[7] Thomas Beck and Herbert Fischer. The if-problem in automatic differentiation. *Journal of Computational and Applied Mathematics*, 50:119–131, 1994.

[8] Claus Bendtsen and Ole Stauning. FADBAD, a flexible C++ package for automatic differentiation — using the forward and backward methods. Technical Report IMM-REP-1996-17, Technical University of Denmark, DK-2800 Lyngby, Denmark, 1996. URL citeseer.nj.nec.com/bendtsen96fadbad.html.

[9] George Biros and Omar Ghattas. Parallel Largrange–Newton–Krylov–Schur methods for PDE-constrained optimization. part i: The Krylov–Schur solver. *SIAM Journal on Scientific Computing*, 27(2):687–713, 2005.

[10] George Biros and Omar Ghattas. Parallel Largrange–Newton–Krylov–Schur methods for PDE-constrained optimization. part ii: The Largrange–Newton solver and its application to optimal control of steady viscous flows. *SIAM Journal on Scientific Computing*, 27(2):687–713, 2005.

[11] C. H. Bischof, L. Roh, and A. J. Mauer-Oats. ADIC: an extensible automatic differentiation tool for ANSI-C. *Software — Practice and Experience*, 27(12):1427–1456, 1997. URL citeseer.nj.nec.com/article/bischof97adic.html.

[12] Alan Carle and Mike Fagan. ADIFOR 3.0 overview. Technical Report CAAM-TR-00-02, Rice University, 2000.

[13] H. S. Chung and J. J. Alonso. Using gradients to construct response surface models for high-dimensional design optimization problems. In *39th AIAA Aerospace Sciences Meeting*, Reno, NV, January 2001. AIAA-2001-0922.

[14] J.E. Dennis and J. J. Moreé. Quasi-Newton methods, motivation and theory. *SIAM Review*, 19(1):46–89, 1977.

[15] Richard P. Dwight and Joël Brezillion. Effect of approximations of the discrete adjoint on gradient-based optimization. *AIAA Journal*, 44(12):3022–3031, 2006.

[16] C. Faure and Y. Papegay. *Odyssée Version 1.6: The Language Reference Manual*. INRIA, 1997. Rapport Technique 211.

[17] Ralf Giering and Thomas Kaminski. Applying TAF to generate efficient derivative code of Fortran 77-95 programs. In *Proceedings of GAMM 2002, Augsburg, Germany*, 2002.

[18] Michael B. Giles and Niles A. Pierce. An introduction to the adjoint approach to design. *Flow, Turbulence and Combustion*, 65:393–415, 2000.

[19] Philip E. Gill, Walter Murray, and Michael A. Saunders. SNOPT: An SQP algorithm for large-scale constrained optimization. *SIAM Review*, 47(1):99–131, 2005. doi:10.1137/S0036144504446096.

[20] Mark S. Gockenbach. Understanding Code Generated by TAMC. IAAA Paper TR00-29, Department of Computational and Applied Mathematics, Rice University, Texas, USA, 2000. URL http://www.math.mtu.edu/~msgocken.

[21] Andreas Griewank. *Evaluating Derivatives*. SIAM, Philadelphia, 2000.

[22] Andreas Griewank, Christian Bischof, George Corliss, Alan Carle, and Karen Williamson. Derivative convergence for iterative equation solvers. *Optimization Methods and Software*, 2: 321–355, 1993.

[23] Andreas Griewank, David Juedes, and Jean Utke. Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software*, 22(2):131–167, 1996. ISSN 0098-3500. URL http://www.acm.org/pubs/citations/journals/toms/1996-22-2/p131-griewank/.

[24] L. Hascoët and V Pascual. Tapenade 2.1 user's guide. Technical report 300, INRIA, 2004. URL http://www.inria.fr/rrrt/rt-0300.html.

[25] E. J. Haug, K. K. Choi, and V. Komkov. *Design Sensitivity Analysis of Structural Systems*, volume 177 of *Mathematics in Science and Engineering*. Academic Press, 1986. URL citeseer.ist.psu.edu/625763.html.

[26] J.E. Hicken and D.W. Zingg. A parallel Newton–Krylov solver for the Euler equations discretized using simultaneous approximation terms. *AIAA Journal*, 46(11), 2008.

[27] R. M. Hicks and P. A. Henne. Wing design by numerical optimization. *Journal of Aircraft*, 15:407–412, 1978.

[28] Kai James, Jorn S. Hansen, and Joaquim R. R. A. Martins. Structural topology optimization for multiple load cases using a dynamic aggregation technique. *Engineering Optimization*, 41 (12):1103–1118, December 2009. doi:10.1080/03052150902926827.

[29] A. Jameson. Aerodynamic design via control theory. *Journal of Scientific Computing*, 3(3): 233–260, sep 1988.

[30] A. Jameson, L. Martinelli, and N. A. Pierce. Optimum aerodynamic design using the Navier–Stokes equations. *Theoretical and Computational Fluid Dynamics*, 10:213–237, 1998.

[31] Graeme J. Kennedy and Joaquim R. R. A. Martins. Parallel solution methods for aerostructural analysis and design optimization. In *Proceedings of the 13th AIAA/ISSMO Multidisciplinary Analysis Optimization Conference*, Forth Worth, TX, September 2010. AIAA 2010-9308.

[32] Gaetan Kenway, Graeme Kennedy, and Joaquim Martins. A scalable parallel approach for high-fidelity aerostructural analysis and optimization. In *53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*. American Institute of Aeronautics and Astronautics, Apr 2012. doi:10.2514/6.2012-1922. URL http://dx.doi.org/10.2514/6.2012-1922.

[33] Edmund Lee and Joaquim R. R. A. Martins. Structural topology optimization with design-dependent pressure loads. *Computer Methods in Applied Mechanics and Engineering*, 2011. (Submitted, Jul 4, 2011).

[34] Edmund Lee, Kai A. James, and Joaquim R. R. A. Martins. Stress-constrained topology optimization with design-dependent loading. *Structural and Multidisciplinary Optimization*, 2011. doi:10.1007/s00158-012-0780-x. (In press)).

[35] Mark Lutz. *Programming Python*. O'Reilly & Associates, Inc., Cambridge, MA 02140, Fall 1996. ISBN 1-56592-197-6. URL http://www.oreilly.com/catalog/python.

[36] Tatyana Luzyanina and Gennady Bocharov. Critical issues in the numerical treatment of the parameter estimation problems in immunology. *Journal of Computational Mathematics*, 30 (1):59–79, jan 2012. doi:10.4208/jcm.1110-m11si12. URL http://dx.doi.org/10.4208/jcm.1110-m11si12.

[37] J. N. Lyness. Numerical algorithms based on the theory of complex variable. In *Proceedings — ACM National Meeting*, pages 125–133, Washington DC, 1967. Thompson Book Co.

[38] J. N. Lyness and C. B. Moler. Numerical differentiation of analytic functions. *SIAM Journal on Numerical Analysis*, 4(2):202–210, 1967. ISSN 0036-1429 (print), 1095-7170 (electronic).

[39] Charles A. Mader and Joaquim R. R. A. Martins. An automatic differentiation discrete adjoint approach for time-spectral computational fluid dynamics. *AIAA Journal*, 2012. (Accepted subject to revisions).

[40] Charles A. Mader and Joaquim R. R. A. Martins. Computation of aircraft stability derivatives using an automatic differentiation adjoint approach. *AIAA Journal*, 49(12):2737–2750, 2011. doi:10.2514/1.55678.

[41] Charles A. Mader, Joaquim R. R. A. Martins, Juan J. Alonso, and Edwin van der Weide. ADjoint: An approach for the rapid development of discrete adjoint solvers. *AIAA Journal*, 46(4):863–873, April 2008. doi:10.2514/1.29123.

[42] Charles Alexander Mader. ADjoint: An approach for the rapid development of discrete adjoint solvers. Master's thesis, University of Toronto Institute for Aerospace Studies, Toronto, ON, 2007.

[43] Joaquim R. R. A. Martins. A guide to the complex-step derivative approximation, 2003. URL http://mdolab.utias.utoronto.ca/resources/complex-step/.

[44] Joaquim R. R. A. Martins. *A Coupled-Adjoint Method for High-Fidelity Aero-Structural Optimization*. PhD thesis, Stanford University, Stanford, CA, 2002.

[45] Joaquim R. R. A. Martins, Juan J. Alonso, and James J. Reuther. Complete configuration aero-structural optimization using a coupled sensitivity analysis method. In *Proceedings of the 9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, Atlanta, GA, September 2002. AIAA 2002-5402.

[46] Joaquim R. R. A. Martins, Peter Sturdza, and Juan J. Alonso. The complex-step derivative approximation. *ACM Transactions on Mathematical Software*, 29(3):245–262, 2003. doi:10.1145/838250.838251.

[47] Joaquim R. R. A. Martins, Juan J. Alonso, and James J. Reuther. High-fidelity aerostructural design optimization of a supersonic business jet. *Journal of Aircraft*, 41(3):523–530, 2004. doi:10.2514/1.11478.

[48] Joaquim R. R. A. Martins, Juan J. Alonso, and James J. Reuther. A coupled-adjoint sensitivity analysis method for high-fidelity aero-structural design. *Optimization and Engineering*, 6(1): 33–62, March 2005. doi:10.1023/B:OPTE.0000048536.47956.62.

[49] Siva Nadarajah and Antony Jameson. A comparison of the continuous and discrete adjoint approach to automatic aerodynamic optimization. In *Proceedings of the 38th AIAA Aerospace Sciences Meeting and Exhibit*, Reno, NV, 2000. AIAA 2000-0667.

[50] Uew Naumann. *The Art of Differentiating Computer Programs — An Introduction to Algorithmic Differentiation*. SIAM, 2011.

[51] James C. Newman III, David L. Whitfield, and W. Kyle Anderson. Step-size independent approach for multidisciplinary sensitivity analysis. *Journal of Aircraft*, 40(3):566–573, 2003.

[52] F. W. J. Olver. *Error Analysis of Complex Arithmetic*, pages 279–292. 1983.

[53] V. Pascual and L. Hascoët. Extension of TAPENADE towards Fortran 95. In H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Tools*, Lecture Notes in Computational Science and Engineering. Springer, 2005.

[54] John D. Pryce and John K. Reid. AD01, a Fortran 90 code for automatic differentiation. Report RAL-TR-1998-057, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, OX11 OQX, U.K., 1998.

[55] J. Reuther, J. J. Alonso, J. C. Vassberg, A. Jameson, and L. Martinelli. An efficient multiblock method for aerodynamic analysis and design on distributed memory systems. *AIAA Paper* 97-1893, 1997.

[56] J. Reuther, J. J. Alonso, J. R. R. A. Martins, and S. C. Smith. A coupled aero-structural optimization method for complete aircraft configurations. In *Proceedings of the 37th AIAA Aerospace Sciences Meeting and Exhibit*, Reno, NV, January 1999. AIAA 99-0187.

[57] James J. Reuther, Antony Jameson, Juan J. Alonso, , Mark J. Rimlinger, and David Saunders. Constrained multipoint aerodynamic shape optimization using an adjoint formulation and parallel computers, part 1. *Journal of Aircraft*, 36(1):51–60, 1999.

[58] Andreas Rhodin. IMAS — Integrated Modeling and Analysis System for the solution of optimal control problems. *Computer Physics Communications*, (107):21–38, 1997.

[59] S. Y. Ruo, J. B. Malone, J. J. Horsten, and R. Houwink. The LANN program — an experimental and theoretical study of steady and unsteady transonic airloads on a supercritical wing. In *Proceedings of the 16th Fluid and PlasmaDynamics Conference*, Danvers, MA, July 1983. AIAA 1983-1686.

[60] A. Saltelli, M. Ratto, T. Andres, F. Campolongo, J. Cariboni, D. Gatelli, M. Saisana, and S. Tarantola. *Global Sensitivity Analysis: The Primer*. John Wiley & Sons Ltd., 2008.

[61] Laura L. Sherman, III Arthur C. Taylor, Larry L. Green, Perry A. Newman, Gene W. Hou, and Vamshi Mohan Korivi. First- and second-order aerodynamic sensitivity derivatives via automatic differentiation with incremental iterative methods. *J. Comput. Phys.*, 129(2):307–331, 1996. ISSN 0021-9991. doi:http://dx.doi.org/10.1006/jcph.1996.0252.

[62] Dmitri Shiriaev. ADOL–F automatic differentiation of Fortran codes. In Martin Berz, Christian H. Bischof, George F. Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 375–384. SIAM, Philadelphia, Penn., 1996. URL citeseer.nj.nec.com/shiriaev96adolf.html.

[63] Ole Sigmund. On the usefulness of non-gradient approaches in topology optimization. *Structural and Multidisciplinary Optimization*, 43:589—596, 2011. doi:10.1007/s00158-011-0638-7.

[64] William Squire and George Trapp. Using complex variables to estimate derivatives of real functions. *SIAM Review*, 40(1):110–112, 1998. ISSN 0036-1445 (print), 1095-7200 (electronic). URL http://epubs.siam.org/sam-bin/dbq/article/31241.

[65] P. Sturdza, V. M. Manning, I. M. Kroo, and R. R. Tracy. Boundary layer calculations for preliminary design of wings in supersonic flow. *AIAA Paper* 99-3104, 1999.

[66] Peter Sturdza. An aerodynamic design method for supersonic natural laminar flow aircraft. PhD thesis 153–159, Stanford University, Stanford, California, 2004.

[67] F. van Keulen, R.T. Haftka, and N.H. Kim. Review of options for structural design sensitivity analysis. part 1: Linear systems. *Computer Methods in Applied Mechanics and Engineering*, 194:3213–3243, 2005.

[68] J. Yao, J. J. Alonso, A. Jameson, and F. Liu. Development and validation of a massively parallel flow solver for turbomachinery flow. *Journal of Propulsion and Power*, 17(3):659–668, 2001.

[69] A. Zole and M. Karpel. Continuous gust response and sensitivity derivatives using state-space models. *Journal of Aircraft*, 31(5):1212 – 1214, 1994.

Chapter 5

Constrained Optimization

5.1 Introduction

Engineering design optimization problems are very rarely unconstrained. Moreover, the constraints that appear in these problems are typically nonlinear. This motivates our interest in general *nonlinearly constrained* optimization theory and methods in this chapter.

Recall the statement of a general optimization problem,

$$\begin{aligned} \text{minimize} \quad & f(x) && (5.1)\\ \text{with respect to} \quad & x \in \mathbb{R}^n && (5.2)\\ \text{subject to} \quad & \hat{c}_j(x) = 0, \quad j = 1, \ldots, \hat{m} && (5.3)\\ & c_k(x) \geq 0, \quad k = 1, \ldots, m && (5.4) \end{aligned}$$

Example 5.24. Graphical Solution of a Constrained Optimization Problem

Suppose we want to solve the following optimization problem,

$$\begin{aligned} \text{minimize} \quad & f(x) = 4x_1^2 - x_1 - x_2 - 2.5 && (5.5)\\ \text{with respect to} \quad & x_1, x_2 && (5.6)\\ \text{subject to} \quad & c_1(x) = x_2^2 - 1.5x_1^2 + 2x_1 - 1 \geq 0, && (5.7)\\ & c_2(x) = x_2^2 + 2x_1^2 - 2x_1 - 4.25 \leq 0 && (5.8) \end{aligned}$$

How can we solve this? One intuitive way would be to make a contour plot of the objective function, overlay the constraints and determine the feasible region, as shown in Fig. 5.1. By inspection, it is easy to see where the constrained optimum is. At the optimum, only one of the constraints is active. However, we want to be able to find the minimum numerically, without having to plot the functions, since this is impractical in the general case. We can see that in this case, the feasible space comprises two disconnected regions. In addition, although all the functions involved are smooth, the boundaries of the feasible regions are nonmsooth. These characteristics complicate the numerical solution of constrained optimization problems.

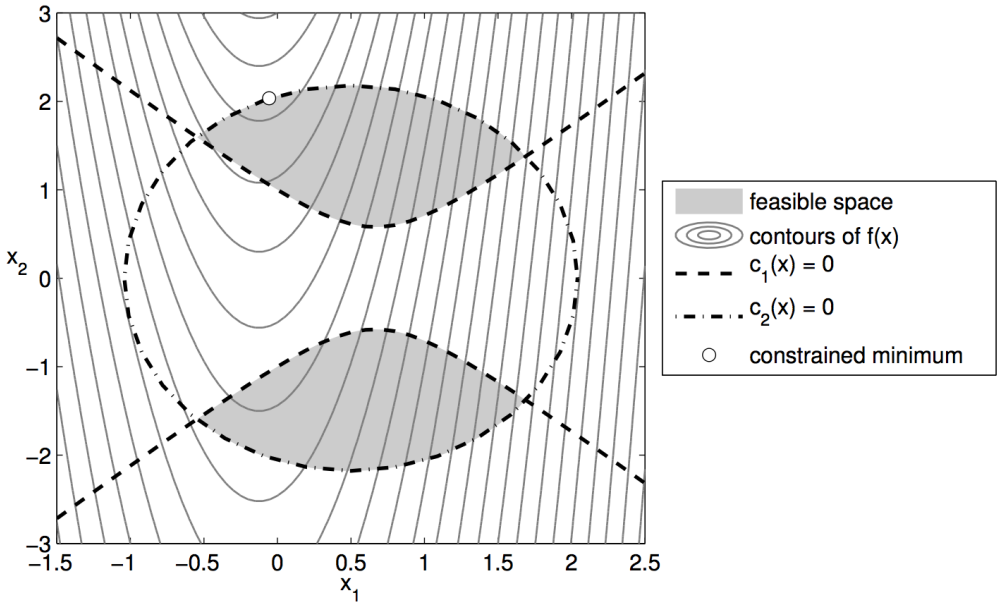


Figure 5.1: Example contours and feasible regions for a simple constrained optimization problem.

5.2 Optimality Conditions for Constrained Problems

The optimality conditions for nonlinearly constrained problems are important because they form the basis of many algorithms for solving such problems.

5.2.1 Nonlinear Equality Constraints

Suppose we have the following optimization problem with equality constraints,

$$\begin{aligned} \text{minimize} \quad & f(x) \\ \text{with respect to} \quad & x \in \mathbb{R}^n \\ \text{subject to} \quad & \hat{c}_j(x) = 0, \quad j = 1, \ldots, \hat{m} \end{aligned}$$

To solve this problem, we could solve for $\hat{m}$ components of x by using the equality constraints to express them in terms of the other components. The result would be an unconstrained problem with $n - \hat{m}$ variables. However, this procedure is only feasible for simple explicit functions.

Joseph Louis Lagrange is credited with developing a more general method to solve this problem, which we now review. At a stationary point, the total differential of the objective function has to be equal to zero, i.e.,

$$\mathrm{d}f = \frac{\partial f}{\partial x_1}\,\mathrm{d}x_1 + \frac{\partial f}{\partial x_2}\,\mathrm{d}x_2 + \cdots + \frac{\partial f}{\partial x_n}\,\mathrm{d}x_n = \nabla f^T\,\mathrm{d}x = 0. \tag{5.9}$$

Unlike unconstrained optimization, the infinitesimal vector $\mathrm{d}x = [\,\mathrm{d}x_1, \mathrm{d}x_2, \ldots, \mathrm{d}x_n]^T$ is not arbitrary here; if x is the sought after minimum, then the perturbation $x + \mathrm{d}x$ must be feasible: $\hat{c}_j(x + \mathrm{d}x) = 0$. Consequently, the above equation *does not* imply that $\nabla f = 0$.

For a feasible point, the total differential of each of the constraints $(\hat{c}_1, \ldots \hat{c}_{\hat{m}})$ must also be zero:

$$\mathrm{d}\hat{c}_j = \frac{\partial \hat{c}_j}{\partial x_1}\,\mathrm{d}x_1 + \cdots + \frac{\partial \hat{c}_j}{\partial x_n}\,\mathrm{d}x_n = \nabla \hat{c}_j^T\,\mathrm{d}x = 0, \quad j = 1, \ldots, \hat{m} \tag{5.10}$$

To interpret the above equation, recall that the gradient of a function is orthogonal to its contours. Thus, since the displacement $\mathrm{d}x$ satisfies $\hat{c}_j(x+dx)=0$ (the equation for a contour), it follow that $\mathrm{d}x$ is orthogonal to the gradient $\nabla\hat{c}_j$.

Lagrange suggested that one could multiply each constraint variation by a scalar $\hat{\lambda}_j$ and subtract it from the objective function variation,

$$\mathrm{d}f - \sum_{j=1}^{\hat{m}} \hat{\lambda}_j \,\mathrm{d}\hat{c}_j = 0 \Rightarrow \sum_{i=1}^{n} \left(\frac{\partial f}{\partial x_i} - \sum_{j=1}^{\hat{m}} \hat{\lambda}_j \frac{\partial \hat{c}_j}{\partial x_i} \right) \mathrm{d}x_i = 0. \tag{5.11}$$

Notice what has happened: the components of the infinitesimal vector $\mathrm{d}x$ have become independent and arbitrary, because we have accounted for the constraints. Thus, for this equation to be satisfied, we need a vector $\hat{\lambda}$ such that the expression inside the parenthesis vanishes, i.e.,

$$\frac{\partial f}{\partial x_i} - \sum_{j=1}^{\hat{m}} \hat{\lambda}_j \frac{\partial \hat{c}_j}{\partial x_i} = 0, \qquad (i = 1, 2, \ldots, n) \tag{5.12}$$

which is a system of n equations and $n+m$ unknowns. To close the system, we recognize that the m constraints must also be satisfied.

Suppose we define a function as the objective function minus a weighted sum of the constraints,

$$\mathcal{L}(x, \hat{\lambda}) = f(x) - \sum_{j=1}^{\hat{m}} \hat{\lambda}_j \hat{c}_j(x) \Rightarrow$$

$$\boxed{\mathcal{L}(x, \hat{\lambda}) = f(x) - \hat{\lambda}^T \hat{c}(x)} \tag{5.13}$$

We call this function the *Lagrangian* of the constrained problem, and the weights the *Lagrange* multipliers. A stationary point of the Lagrangian with respect to both x and $\hat{\lambda}$ will satisfy

$$\frac{\partial \mathcal{L}}{\partial x_i} = \frac{\partial f}{\partial x_i} - \sum_{j=1}^{\hat{m}} \hat{\lambda}_j \frac{\partial \hat{c}_j}{\partial x_i} = 0, \quad (i = 1, \ldots, n) \tag{5.14}$$

$$\frac{\partial \mathcal{L}}{\partial \hat{\lambda}_j} = \hat{c}_j = 0, \quad (j = 1, \ldots, \hat{m}). \tag{5.15}$$

Thus, a stationary point of the Lagrangian encapsulates our required conditions: the constraints are satisfied and the gradient conditions (5.12) are satisfied. These first-order conditions are known as the *Karush–Kuhn–Tucker (KKT)* conditions. They are necessary conditions for the optimum of a constrained problem.

As in the unconstrained case, the first-order conditions are not sufficient to guarrantee a local minimum. For this, we turn to the second-order sufficient conditions (which, as in the unconstrained case, are not necessary). For equality constrained problems we are concerned with the behaviour of the Hessian of the Lagrangian, denoted $\nabla^2_{xx}\mathcal{L}(x, \hat{\lambda})$, at locations where the KKT conditions hold. In particular, we look for positive-definiteness *in a subspace defined by the linearized constraints.* Geometrically, if we move away from a stationary point $(x^*, \hat{\lambda}^*)$ along a direction w that satisfies the linearized constraints, the Lagrangian should look like a quadratric along this direction. To be precise, the second-order sufficient conditions are

$$w^T \nabla^2_{xx}\mathcal{L}(x^*, \hat{\lambda}^*) w > 0,$$

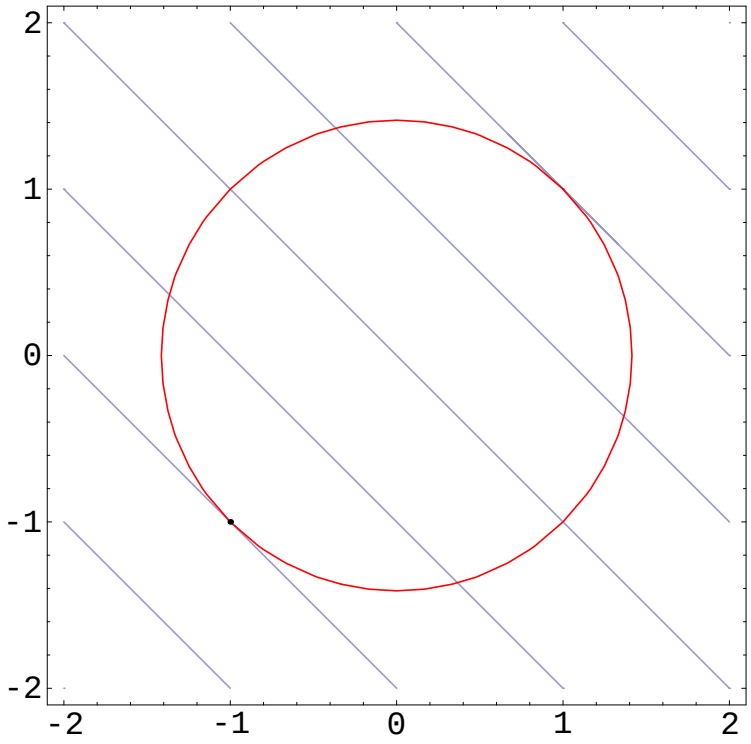


Figure 5.2: Contour plot for constrained problem (5.18).

for all $w \in \mathbb{R}^n$ such that

$$\nabla \hat{c}_j(x^*)^T w = 0, \qquad j = 1, \ldots, \hat{m}.$$

Example 5.25. Problem with Single Equality Constraint

Consider the following equality constrained problem:

$$\begin{aligned} \text{minimize} \quad & f(x) = x_1 + x_2 && (5.16)\\ \text{weight respect to} \quad & x_1, x_2 && (5.17)\\ \text{subject to} \quad & \hat{c}_1(x) = x_1^2 + x_2^2 - 2 = 0 && (5.18) \end{aligned}$$

The objective function and constraint of the above problem are shown in Fig. 5.2. By inspection we can see that the feasible region for this problem is a circle of radius $\sqrt{2}$. The solution x^* is obviously $(-1, -1)^T$. From any other point in the circle it is easy to find a way to move in the feasible region (the boundary of the circle) while decreasing f.

In this example, the Lagrangian is

$$\mathcal{L} = x_1 + x_2 - \hat{\lambda}_1(x_1^2 + x_2^2 - 2) \tag{5.19}$$

And the optimality conditions are

$$\begin{aligned} \nabla_x \mathcal{L} &= \begin{bmatrix} 1 - 2\hat{\lambda}_1 x_1 \\ 1 - 2\hat{\lambda}_1 x_2 \end{bmatrix} = 0 \Rightarrow \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \frac{1}{2\hat{\lambda}_1} \\ \frac{1}{2\hat{\lambda}_1} \end{bmatrix} \\ \nabla_{\hat{\lambda}_1} \mathcal{L} &= x_1^2 + x_2^2 - 2 = 0 \Rightarrow \hat{\lambda}_1 = \pm\frac{1}{2} \end{aligned}$$

To establish which are minima as opposed to other types of stationary points, we need to look at the second-order conditions. Directions $w = (w_1, w_2)^T$ that satisfy the linearized constraints are given by

$$\begin{aligned} \nabla \hat{c}_1(x^*)^T w &= \frac{1}{\hat{\lambda}_1}(w_1 + w_2) = 0 \\ \Rightarrow \quad w_2 &= -w_1 \end{aligned}$$

while the Hessian of the Lagrangian at the stationary points is

$$\nabla^2_x \mathcal{L} = \begin{bmatrix} -2\hat{\lambda}_1 & 0 \\ 0 & -2\hat{\lambda}_1 \end{bmatrix}.$$

Consequently, the Hessian of the Lagrangian in the subspace defined by w is

$$w^T \nabla^2_{xx} \mathcal{L}(x^*) w = \begin{bmatrix} w_1 & -w_1 \end{bmatrix} \begin{bmatrix} -2\hat{\lambda}_1 & 0 \\ 0 & -2\hat{\lambda}_1 \end{bmatrix} \begin{bmatrix} w_1 \\ -w_1 \end{bmatrix} = -4\hat{\lambda}_1 w_1^2$$

In this case $\hat{\lambda}_1^* = -\frac{1}{2}$ corresponds to a positive-definite Hessian (in the space w) and, therefore, the solution to the problem is $(x_1, x_2)^T = (\frac{1}{2\hat{\lambda}_1}, \frac{1}{2\hat{\lambda}_1})^T = (-1, -1)^T$.

Also note that at the solution the constraint normal $\nabla \hat{c}_1(x^*)$ is parallel to $\nabla f(x^*)$, i.e., there is a scalar $\hat{\lambda}_1^*$ such that

$$\nabla f(x^*) = \hat{\lambda}_1^* \nabla \hat{c}_1(x^*). \tag{5.20}$$

5.2.2 Alternative Derivation of Lagrangian

We can derive the Lagrangian expression by examining the first-order Taylor series approximations to the objective and constraint functions. To retain feasibility with respect to $\hat{c}_1(x) = 0$ we require that

$$\begin{aligned} \hat{c}_1(x + d) &= 0 \Rightarrow \\ \hat{c}_1(x + d) &= \underbrace{\hat{c}_1(x)}_{=0} + \nabla \hat{c}_1^T(x) d + \mathcal{O}(d^T d). \end{aligned}$$

Linearizing this we get,

$$\boxed{\nabla \hat{c}_1^T(x) d = 0}.$$

We also know that a direction of improvement must result in a decrease in f, i.e.,

$$f(x + d) - f(x) < 0.$$

Thus to first order we require that

$$\begin{aligned} f(x) + \nabla f^T(x) d - f(x) &< 0 \Rightarrow \\ \boxed{\nabla f^T(x) d < 0}&. \end{aligned}$$

A necessary condition for optimality is that there be no direction satisfying both of these conditions. The only way that such a direction cannot exist is if $\nabla f(x)$ and $\nabla \hat{c}_1(x)$ are parallel, that is, if $\nabla f(x) = \hat{\lambda}_1 \nabla \hat{c}_1(x)$ holds.

By defining the Lagrangian function

$$\mathcal{L}(x, \hat{\lambda}_1) = f(x) - \hat{\lambda}_1 \hat{c}_1(x), \tag{5.21}$$

and noting that $\nabla_x \mathcal{L}(x, \hat{\lambda}_1) = \nabla f(x) - \hat{\lambda}_1 \nabla \hat{c}_1(x)$, we can state the necessary optimality condition as follows: At the solution x^* there is a scalar $\hat{\lambda}_1^*$ such that $\nabla_x \mathcal{L}(x^*, \hat{\lambda}_1^*) = 0$.

Thus we can search for solutions of the equality-constrained problem by searching for a stationary point of the Lagrangian function. The scalar $\hat{\lambda}_1$ is the Lagrange multiplier for the constraint $\hat{c}_1(x) = 0$.

5.2.3 Nonlinear Inequality Constraints

Suppose we now have a general problem with equality and inequality constraints.

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{w.r.t} & x \in \mathbb{R}^n \\ \text{subject to} & \hat{c}_j(x) = 0, \quad j = 1, \ldots, \hat{m} \\ & c_k(x) \geq 0, \quad k = 1, \ldots, m \end{array}$$

We can apply the same Lagrangian formulation if we can transform our inequality constraints into equality constraints. We do this by introducing additional variables called slack variables. The modification results in a the Lagrangian defined as

$$\mathcal{L}(x, \hat{\lambda}, \lambda, s) = f(x) - \hat{\lambda}^T \hat{c}(x) - \lambda^T \left(c(x) - s^2 \right), \tag{5.22}$$

where λ are the Lagrange multipliers associated with the inequality constraints and s is a vector of *slack variables*. Note that the equality constraint $c(x) - s^2 = 0$ is equivalent to the original inequality constraint

$$\begin{aligned} c(x) - s^2 &= 0 \\ c(x) &= s^2 \\ \Rightarrow c(x) &\geq 0 \end{aligned} \tag{5.23}$$

We can now solve for the first order KKT conditions by taking the partial derivatives with respect to all of the variables:

$$\begin{array}{rcl} \nabla_x \mathcal{L} = 0 \Rightarrow \dfrac{\partial \mathcal{L}}{\partial x_i} & = & \dfrac{\partial f}{\partial x_i} - \displaystyle\sum_{j=1}^{\hat{m}} \hat{\lambda}_j \frac{\partial \hat{c}_j}{\partial x_i} - \sum_{k=1}^{m} \lambda_k \frac{\partial c_k}{\partial x_i} = 0, \quad i = 1, \ldots, n \\ \nabla_{\hat{\lambda}} \mathcal{L} = 0 \Rightarrow \dfrac{\partial \mathcal{L}}{\partial \hat{\lambda}_j} & = & \hat{c}_j = 0, \quad j = 1, \ldots, \hat{m} \\ \nabla_{\lambda} \mathcal{L} = 0 \Rightarrow \dfrac{\partial \mathcal{L}}{\partial \lambda_k} & = & c_k - s_k^2 = 0 \quad k = 1, \ldots, m \\ \nabla_s \mathcal{L} = 0 \Rightarrow \dfrac{\partial \mathcal{L}}{\partial s_k} & = & \lambda_k s_k = 0, \quad k = 1, \ldots, m \\ \lambda_k & \geq & 0, \quad k = 1, \ldots, m. \end{array}$$

This results in $n + \hat{m} + 2m$ equations.

The equations $\lambda_k s_k = 0$ are called the complementarity conditions are each equation has two possibilities:

$s_k > 0$: which means that the k-th constraint is inactive, and thus the corresponding Lagrange multiplier must be zero ($\lambda_k = 0$).

$s_k = 0$: the k-th constraint is active. λ_k must then be non-negative, otherwise from the first equations, the gradient of objective and gradient of constraint point in the same direction.

Sufficient conditions are obtained by examining the second-order requirements. The set of sufficient conditions is as follows:

1. KKT necessary conditions must be satisfied at x^*.
2. The Hessian matrix of the Lagrangian,

$$\nabla^2 \mathcal{L} = \nabla^2 f(x^*) - \sum_{j=1}^{\hat{m}} \hat{\lambda}_j \nabla^2 \hat{c}_j - \sum_{k=1}^{m} \lambda_k \nabla^2 c_k \tag{5.24}$$

is positive definite in the feasible space. This is a subspace of n-space and is defined as follows: any direction y that satisfies

$$y \neq 0 \tag{5.25}$$
$$\nabla \hat{c}_j^T(x^*) y = 0, \quad \text{for all} \quad j = 1, \ldots, \hat{m} \tag{5.26}$$
$$\nabla c_k^T(x^*) y = 0, \quad \text{for all} \quad k \quad \text{for which} \quad \lambda_k > 0. \tag{5.27}$$

Then the Hessian of the Lagrangian in feasible space must be positive definite,

$$y^T \nabla^2 \mathcal{L}(x^*) y > 0. \tag{5.28}$$

Example 5.26. Problem with a Single Inequality Constraint

Suppose we now have the same problem, but with an inequality replacing the equality constraint,

$$\begin{aligned} \text{minimize } f(x) &= x_1 + x_2 \\ \text{s.t.} \quad c_1(x) &= 2 - x_1^2 - x_2^2 \geq 0 \end{aligned}$$

The feasible region is now the circle and its interior. Note that $\nabla c_1(x)$ now points towards the center of the circle. Graphically, we can see that the solution is still $(-1, -1)^T$ and therefore $\lambda_1^* = 1/2$.

Given a point x that is not optimal, we can find a step d that both stays feasible and decreases the objective function f, to first order. As in the equality constrained case, the latter condition is expressed as

$$\boxed{\nabla f^T(x) d < 0}. \tag{5.29}$$

The first condition, however is slightly different, since the constraint is not necessarily zero, i.e.

$$c_1(x + d) \geq 0 \tag{5.30}$$

Performing a Taylor series expansion we have,

$$\underbrace{c_1(x + d)}_{\geq 0} \approx c_1(x) + \nabla c_1^T(x) d. \tag{5.31}$$

Thus feasibility is retained to a first order if

$$\boxed{c_1(x) + \nabla c_1^T(x) d \geq 0}. \tag{5.32}$$

In order to find valid steps d it helps two consider two possibilities.

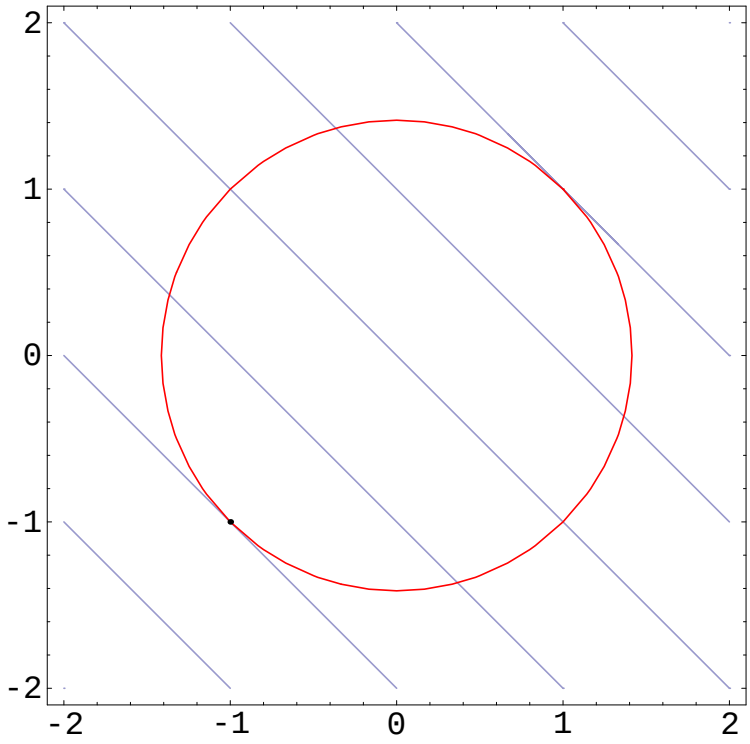


Figure 5.3

1. Suppose x lies strictly inside the circle ($c_1(x) > 0$). In this case, any vector d satisfies the condition (5.32), provided that its length is sufficiently small. The only situation that will prevent us from finding a descent direction is if $\nabla f(x) = 0$. Or in other words, the Lagrangian still applies as long as $\lambda_1 = 0$.

2. Consider now the case in which x lies on the boundary, i.e., $c_1(x) = 0$. The conditions thus become $\nabla f^T(x)d < 0$ and $\nabla c_1^T(x)d \geq 0$. The two regions defined by these conditions fail to intersect only when $\nabla f(x)$ and $\nabla c_1(x)$ point in the same direction, that is, when

$$\nabla f(x)^T d = \lambda_1 c_1(x), \qquad \text{for some } \lambda_1 \geq 0. \tag{5.33}$$

The optimality conditions for these two cases can again be summarized by using the Lagrangian function, that is,

$$\nabla_x \mathcal{L}(x^*, \lambda_1^*) = 0, \qquad \text{for some } \lambda_1^* \geq 0 \qquad \text{and } \lambda_1^* s_1^* = 0. \tag{5.34}$$

The last condition is known as a *complementarity condition* and implies that the Lagrange multiplier can be strictly positive only when the constraint is active.

Example 5.27. Lagrangian Whose Hessian is Not Positive Definite

$$\begin{aligned} \text{minimize} \quad & f(x) = -x_1 x_2 && (5.35)\\ \text{s.t.} \quad & \hat{c}_1(x) = 2 - x_1^2 - x_2^2 = 0 && (5.36)\\ & x_1 \geq 0, \quad x_2 \geq 0 && (5.37) \end{aligned}$$

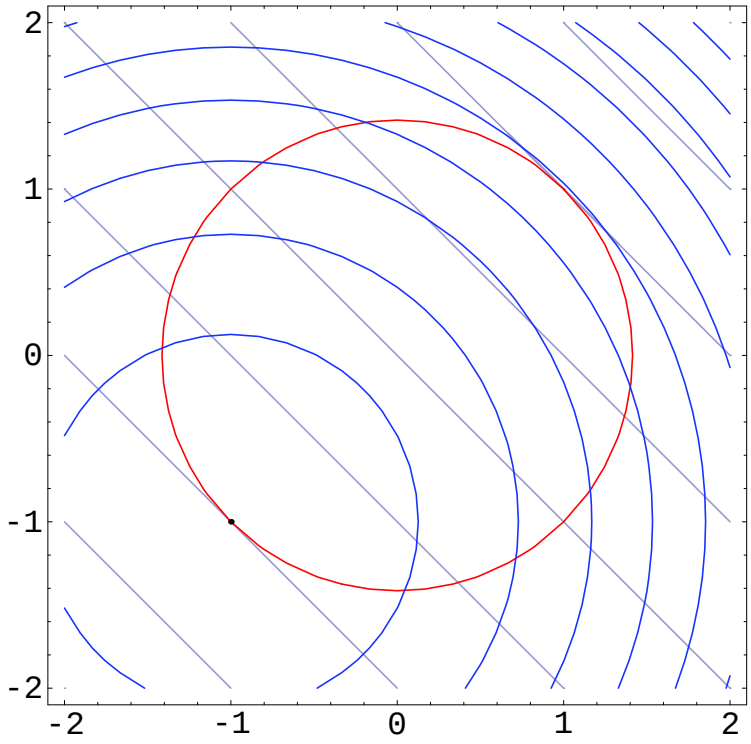


Figure 5.4: Contour plots of $f(x)$, $c_1 = 0$ and $\mathcal{L}(x, \lambda_1^*, s_1^*)$

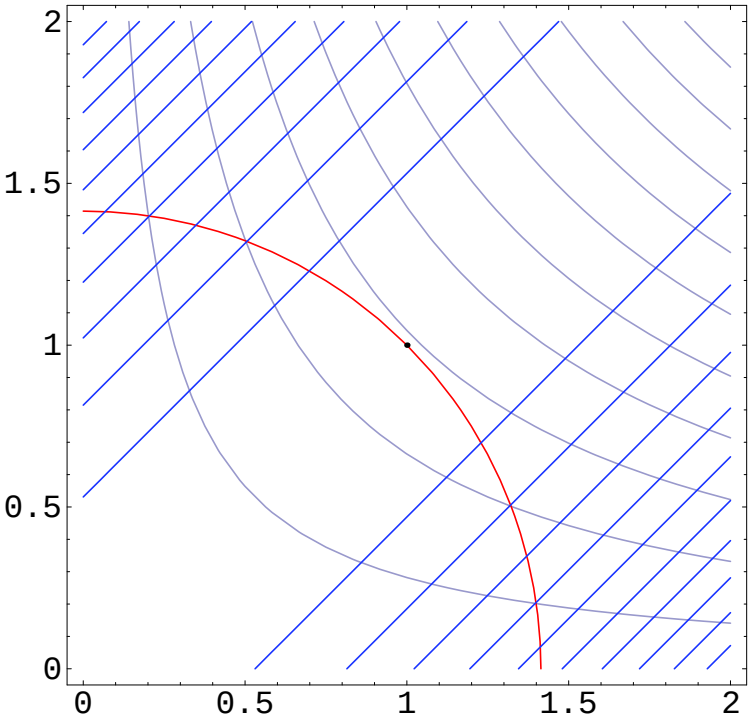


Figure 5.5: Contour plots of $f(x)$, $c_1 = 0$ and $\mathcal{L}(x, \lambda_1^*)$

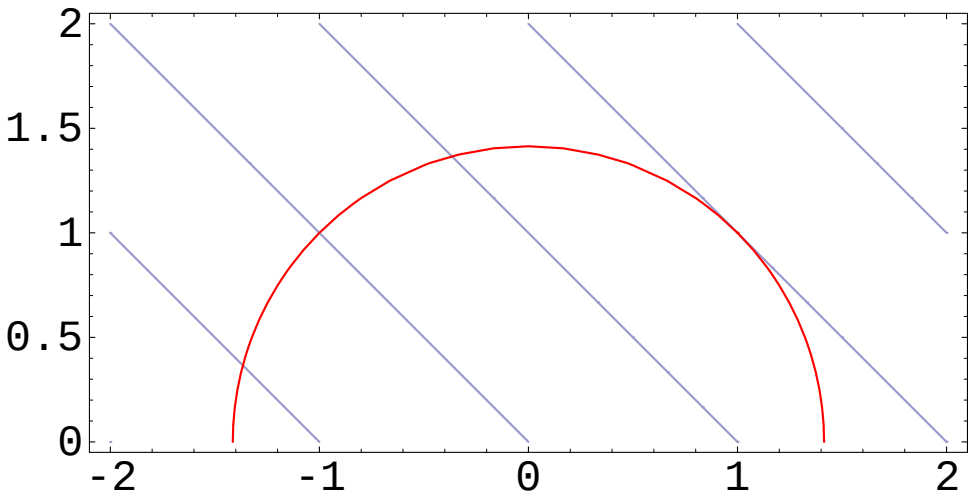


Figure 5.6: Objective and constraint contours for Example 28

If we solve this problem, we will find the Hessian of the Lagrangian shown in Fig. 5.5, which is positive semi-definite. However, it is positive definite in the feasible directions.

Example 5.28. Problem with Two Inequality Constraints

Suppose we now add another inequality constraint,

$$\begin{aligned} \text{minimize} \quad & f(x) = x_1 + x_2 && (5.38)\\ \text{s.t.} \quad & c_1(x) = 2 - x_1^2 - x_2^2 \geq 0, && (5.39)\\ & c_2(x) = x_2 \geq 0. && (5.40) \end{aligned}$$

The feasible region is now a half disk (Fig. 5.6). Graphically, we can see that the solution is now $(-\sqrt{2}, 0)^T$ and that both constraints are active at this point.

The Lagrangian for this problem is

$$\mathcal{L}(x, \lambda, s) = f(x) - \lambda_1 \left(c_1(x) - s_1^2\right) - \lambda_2 \left(c_2(x) - s_2^2\right), \tag{5.41}$$

where $\lambda = (\lambda_1, \lambda_2)^T$ is the vector of Lagrange multipliers. The first order optimality conditions are thus,

$$\nabla_x \mathcal{L}(x^*, \lambda^*) = 0, \qquad \text{for some } \lambda^* \geq 0. \tag{5.42}$$

Applying the complementarity conditions to both inequality constraints,

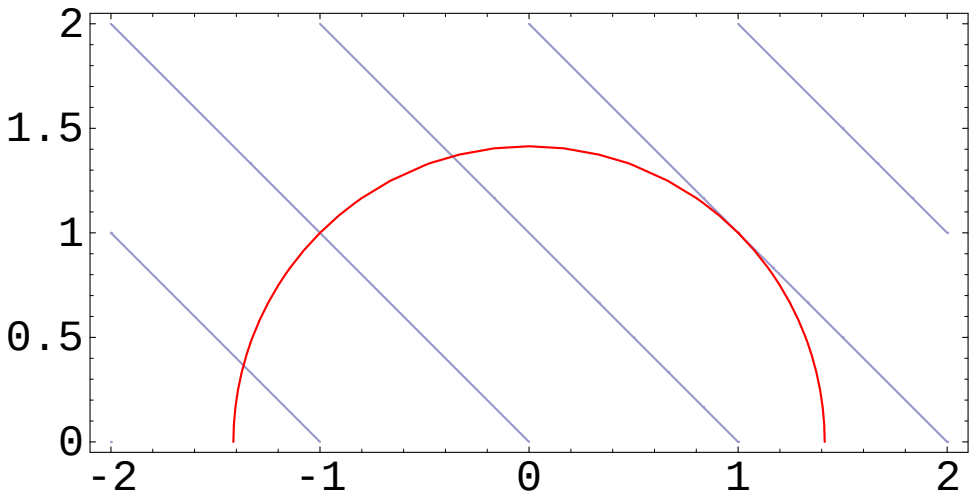
$$\lambda_1^* s_1^* = 0, \qquad \text{and } \lambda_2^* s_2^* = 0. \tag{5.43}$$

For $x^* = (-\sqrt{2}, 0)^T$ we have,

$$\nabla f(x^*) = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \qquad \nabla c_1(x^*) = \begin{bmatrix} 2\sqrt{2} \\ 0 \end{bmatrix}, \qquad \nabla c_2(x^*) = \begin{bmatrix} 0 \\ 1 \end{bmatrix},$$

and $\nabla_x \mathcal{L}(x^*, \lambda^*) = 0$ when

$$\lambda^* = \begin{bmatrix} \frac{1}{2\sqrt{2}} \\ 1 \end{bmatrix}.$$



Now lets consider other feasible points that are not optimal and examine the Lagrangian and its gradients at these points.

For point $x = (\sqrt{2}, 0)^T$, both constraints are again active. However, $\nabla f(x)$ no longer lies in the quadrant defined by $\nabla c_i(x)^T d \geq 0, \quad i = 1, 2$ and therefore there are descent directions that are feasible, like for example $d = (-1, 0)^T$.

Note that $\nabla_x \mathcal{L}(x^*, \lambda^*) = 0$ at this point for $\lambda = (-\frac{1}{2\sqrt{2}}, 1)^T$. However, since λ_1 is negative, the first order conditions are not satisfied at this point.

Now consider the point $x = (1, 0)^T$, for which only the second constraint is active. Linearizing f and c as before, d must satisfy the following to be a feasible descent direction,

$$\begin{aligned} c_1(x+d) &\geq 0 \Rightarrow 1 + \nabla c_1(x)^T d \geq 0, && (5.44) \\ c_2(x+d) &\geq 0 \Rightarrow \nabla c_2(x)^T d \geq 0, && (5.45) \\ f(x+d) - f(x) &< 0 \Rightarrow 1 + \nabla f(x)^T d < 0. && (5.46) \end{aligned}$$

We only need to worry about the last two conditions, since the first is always satisfied for a small enough step.

By noting that

$$\nabla f(x^*) = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \qquad \nabla c_2(x^*) = \begin{bmatrix} 0 \\ 1 \end{bmatrix},$$

we can see that the vector $d = \left(-\frac{1}{2}, \frac{1}{4}\right)$, for example satisfies the two conditions.

Since $c_1(x) > 0$, we must have $\lambda_1 = 0$. In order to satisfy $\nabla_x \mathcal{L}(x, \lambda) = 0$ we would have to find λ_2 such that $\nabla f(x) = \lambda_2 \nabla c_2(x)$. No such λ_2 exists and this point is therefore not an optimum.

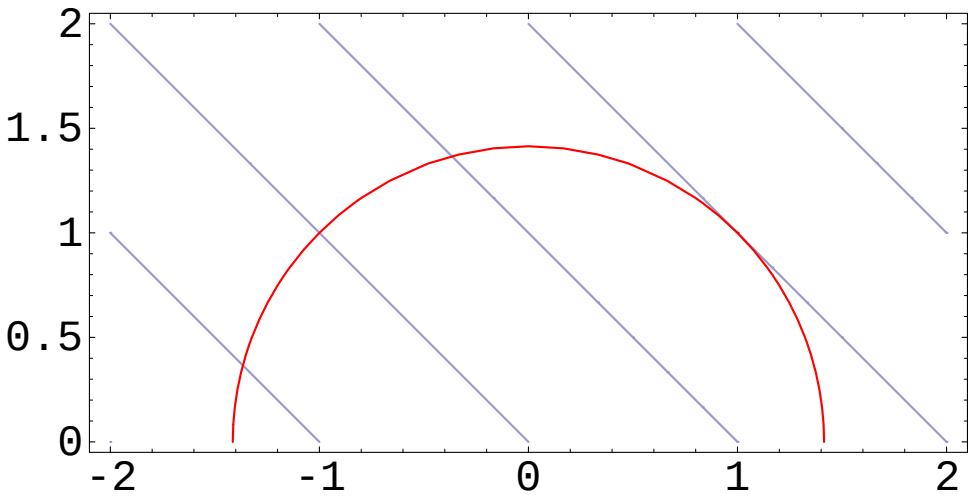
5.2.4 Constraint Qualification

The KKT conditions are derived using certain assumptions and depending on the problem, these assumptions might not hold.

A point x satisfying a set of constraints is a *regular point* if the gradient vectors of the *active* constraints, $\nabla c_j(x)$ are linearly independent.

To illustrate this, suppose we replaced the $\hat{c}_1(x)$ in the first example in this chapter by the equivalent condition

$$\hat{c}_1(x) = \left(x_1^2 + x_2^2 - 2\right)^2 = 0. \qquad (5.47)$$



Then we have

$$\nabla \hat{c}_1(x) = \begin{bmatrix} 4(x_1^2 + x_2^2 - 2)x_1 \\ 4(x_1^2 + x_2^2 - 2)x_2 \end{bmatrix}, \tag{5.48}$$

so $\nabla \hat{c}_1(x) = 0$ for all feasible points and $\nabla f(x) = \hat{\lambda}_1 \nabla \hat{c}_1(x)$ cannot be satisfied. In other words, there is no (finite) Lagrange multiplier that makes the objective gradient parallel to the constraint gradient, so we cannot solve the optimality conditions. This does not imply there is no solution; on the contrary, the solution remains unchanged for the earlier example. Instead, what it means is that most algorithms will fail, because they assume the constraints are linearly independent.

5.3 Penalty Function Methods

One of the ways of solving constrained optimization problems, at least approximately, is by adding a penalty function to the objective function that depends — in some logical way — on the value of the constraints.

The idea is to minimize a sequence of unconstrained minimization problems where the infeasibility of the constraints is minimized together with the objective function.

There two main types of penalization methods: *exterior penalty functions*, which impose a penalty for violation of constraints, and *interior penalty functions*, which impose a penalty for approaching the boundary of an inequality constraint.

5.3.1 Exterior Penalty Functions

The modified objective function is defined as the original one plus a term for each constraint, which is positive when the current point violates the constraint and zero otherwise.

Consider the equality-constrained problem:

$$\begin{aligned} \text{minimize} \quad & f(x) \\ \text{subject to} \quad & \hat{c}(x) = 0 \end{aligned}$$

where $\hat{c}(x)$ is an $\hat{m}$-dimensional vector whose j-th component is $\hat{c}_j(x)$. We assume that all functions are twice continuously differentiable.

We require a penalty for constraint violation to be a continuous function ϕ with the following properties

$$\begin{aligned} \phi(x) &= 0 \quad \text{if } x \text{ is feasible} \\ \phi(x) &> 0 \quad \text{otherwise,} \end{aligned}$$

The new objective function is

$$\pi(x;\rho) = f(x) + \rho\phi(x), \tag{5.49}$$

were ρ is positive and is called the penalty parameter.

The penalty method consists of solving a sequence of unconstrained minimization problems of the form

$$\begin{aligned} \text{minimize} \quad & \pi\left(x;\rho_k\right) \\ \text{w.r.t.} \quad & x \end{aligned}$$

for an increasing sequence of positive values of ρ_k tending to infinity. In general, for finite values of ρ_k, the minimizer of the penalty function violate the equality constraints. The increasing penalty forces the minimizer toward the feasible region.

The steps for this method are listed in Algorithm 11

Algorithm 11 General algorithm

Check termination conditions. if x_k satisfies the optimality conditions, the algorithm terminates successfully.

Minimize the penalty function. With x_k as the starting point, execute an algorithm to solve the unconstrained subproblem

$$\begin{aligned} \text{minimize} \quad & \pi(x;\rho_k) \\ \text{w.r.t.} \quad & x \end{aligned}$$

and let the solution of this subproblem be x_{k+1}.

Increase the penalty parameter. Set ρ_{k+1} to a larger value than ρ_k, set $k = k+1$ and return to 1.

The increase in the penalty parameter for each iteration can range from modest ($\rho_{k+1} = 1.4\rho_k$), to ambitious ($\rho_{k+1} = 10\rho_k$), depending on the problem.

The Quadratic Penalty Method

The quadratic penalty function is defined as

$$\pi(x;\rho) = f(x) + \frac{\rho}{2}\sum_{i=1}^{\hat{m}} \hat{c}_i(x)^2 = f(x) + \frac{\rho}{2}\hat{c}(x)^T\hat{c}(x). \tag{5.50}$$

The penalty is equal to the sum of the square of all the constraints and is therefore greater than zero when any constraint is violated and is zero when the point is feasible.

We can modify this method to handle inequality constraints by defining the penalty for these constraints as

$$\phi(x;\rho) = \frac{\rho}{2}\sum_{i=1}^{m} \left(\max\left[0, -c_i(x)\right]\right)^2. \tag{5.51}$$

Penalty functions suffer from problems of ill conditioning. The solution of the modified problem approaches the true solution as $\lim_{\rho \to +\infty} x^*(\rho) = x^*$, but, as the penalty parameter increases, the condition number of the Hessian matrix of $\pi(x, \rho)$ increases and tends to ∞. This makes the problem increasingly difficult to solve numerically.

The other challenge with penalty methods is that the starting value of the penalty parameter must also be chosen sufficiently large, or else the penalty function could be unbounded from below even though the original constrained problem was not. As an example consider

$$\begin{aligned} \text{minimize} \quad & -2x_1^2 + x_2^2 \\ \text{subject to} \quad & x_1 = 1 \end{aligned} \tag{5.52}$$

The solution to this problem is $x = (1, 0)$. However, the corresponding quadratic penalty function

$$\begin{aligned} \pi(x; \rho) &= -2x_1^2 + x_2^2 + \frac{\rho}{2}(x_1 - 1)^2 \\ &= (-2 + \rho/2)x_1^2 + x_2^2 + \frac{\rho}{2}(-2x_1 + 1) \end{aligned} \tag{5.53}$$

is unbounded below if $\rho < 4$. Safeguards are necessary to detect divergence and increase the penalty parameter.

The Augmented Lagrangian Method

An improvement to the quadratic penalty method is the augmented Lagrangian function:

$$\mathcal{L}_A(x, \lambda; \rho) = f(x) - \sum_i \lambda_i c_i(x) + \frac{\rho}{2} \sum_i c_i(x)^2 \tag{5.54}$$

This is a combination of the standard Lagrangian with a quadratic penalty term.

The optimality condition for this problem is

$$\nabla_x \mathcal{L}_A = \nabla f(x) - \sum_i (\lambda_i - \rho c_i(x)) \nabla c_i(x) \tag{5.55}$$

If we compare this to the optimality conditions for the standard Lagrangian (Eq. (5.20)) we have

$$\lambda_i^* \approx \lambda_i - \rho c_i(x) \tag{5.56}$$

and from this we can deduce an update rule on the Lagrange multipliers.

$$\lambda^{k+1} = \lambda_i^k - \rho^k c_i(x^k) \tag{5.57}$$

Thus, given a starting point for λ_0 we should expect our estimate for the Lagrange multipliers to improve at each iteration. We also note from Eq. (5.56) that

$$c_i(x^k) \approx -\frac{1}{\rho^k}(\lambda_i^* - \lambda_i^k) \tag{5.58}$$

This suggests that either we can reduce the error in our constraints by increasing the penalty parameter ρ, or we can reduce the error by providing Lagrange multipliers that are closer to the optimal Lagrange multipliers. In effect, the Augmented Lagrangian method gives us a method to obtain a good solution for x^* while avoiding the ill-conditioning issues the quadratic penalty method faces. By updating the Lagrange multiplier estimates at every iteration (after each solution of the unconstrained subproblem), we can obtain more accurate solutions without having to increase ρ excessively.

5.3.2 Interior Penalty Methods

Exterior penalty methods generate infeasible points and are therefore not suitable when feasibility has to be strictly maintained. This might be the case if the objective function is undefined or ill-defined outside the feasible region.

The method is analogous to the external penalty method: it creates a sequence of unconstrained modified differentiable functions whose unconstrained minima converge to the optimum solution of the constrained problem in the limit.

Consider the inequality-constrained problem:

$$\begin{aligned} \text{minimize} \quad & f(x) && (5.59)\\ \text{subject to} \quad & c(x) \geq 0 && (5.60) \end{aligned}$$

where $c(x)$ is an m-dimensional vector whose j-th component is $c_j(x)$. Again, we assume that all functions are twice continuously differentiable.

The Logarithmic Barrier Method

The logarithmic barrier function adds a penalty that tends to infinity as x approaches infeasibility. The function is defined as

$$\pi(x, \mu) = f(x) - \mu \sum_{j=1}^{m} \log\left(c_j(x)\right), \tag{5.61}$$

where the positive scalar μ is called the *barrier parameter*.

This is how early barrier methods worked, and the approach is still useful for conceptualization, but modern interior-point methods are no longer solved in this manner. The barrier approach requires that all iterates be strictly feasible (including the starting point), otherwise the logarithm is not real-valued. Instead, slack variables are used so that the iterates may be infeasible.

$$\begin{aligned} \text{minimize} \quad & f(x) - \mu \sum_{j=1}^{m} \log\left(s_j\right) \\ \text{subject to} \quad & c_j(x) - s_j = 0 \\ & s_j \geq 0 \quad \text{automatically enforced through the logarithm} \end{aligned} \tag{5.62}$$

Modern interior-point methods rival SQP methods (discussed in the next section) as being the most effective approaches for nonlinear constrained programming problems.

The Inverse Barrier Function

The inverse barrier function is defined as

$$\pi(x, \mu) = f(x) + \mu \sum_{j=1}^{m} \frac{1}{c_j(x)}, \tag{5.63}$$

and shares many of the same characteristics of the logarithmic barrier.

The solution of the modified problem for both functions approach the real solution as

$$\lim_{\mu \to 0} x^*(\mu) = x^*$$

. Again, the Hessian matrix becomes increasingly ill conditioned as μ approaches zero. Note that unlike exterior penalty methods where the penalty multiplier is increased every iteration, for interior methods the penalty should be decreased every iteration.

Similarly to the an exterior point method, an algorithm using these barrier functions finds the minimum of $\pi(x, \mu_k)$, for a given starting (feasible) point and terminates when norm of gradient is close to zero.

The algorithm then chooses a new barrier parameter μ_{k+1} and a new starting point, finds the minimum of the new problem and so on. A value of 0.1 for the ratio μ_{k+1}/μ_k is usually considered ambitious.

Example 5.29. Constrained Minimization Using a Quadratic Penalty Function

Consider the inequality constrained problem 5.8, introduced in the beginning of this chapter. Fig. 5.7 shows four different subproblems corresponding to increasing penalty parameters ρ. Recall, that this problem has two disjoint feasible regions, and has two local minima. For the lowest value of ρ, we obtain a function whose minimum is close to the global minimum. Although the optimization starts in the feasible region that has the local minimum, the solution of this first subproblem takes the solution past the local minimum towards the global minimum. As ρ increases, the other minimum appears, but by then we already are near the global minimum. As ρ increases further, the solution of each subproblem approaches the correct constrained minimum, but the penalized function becomes highly nonlinear, as can be seen from the abrupt change in the contour spacing.

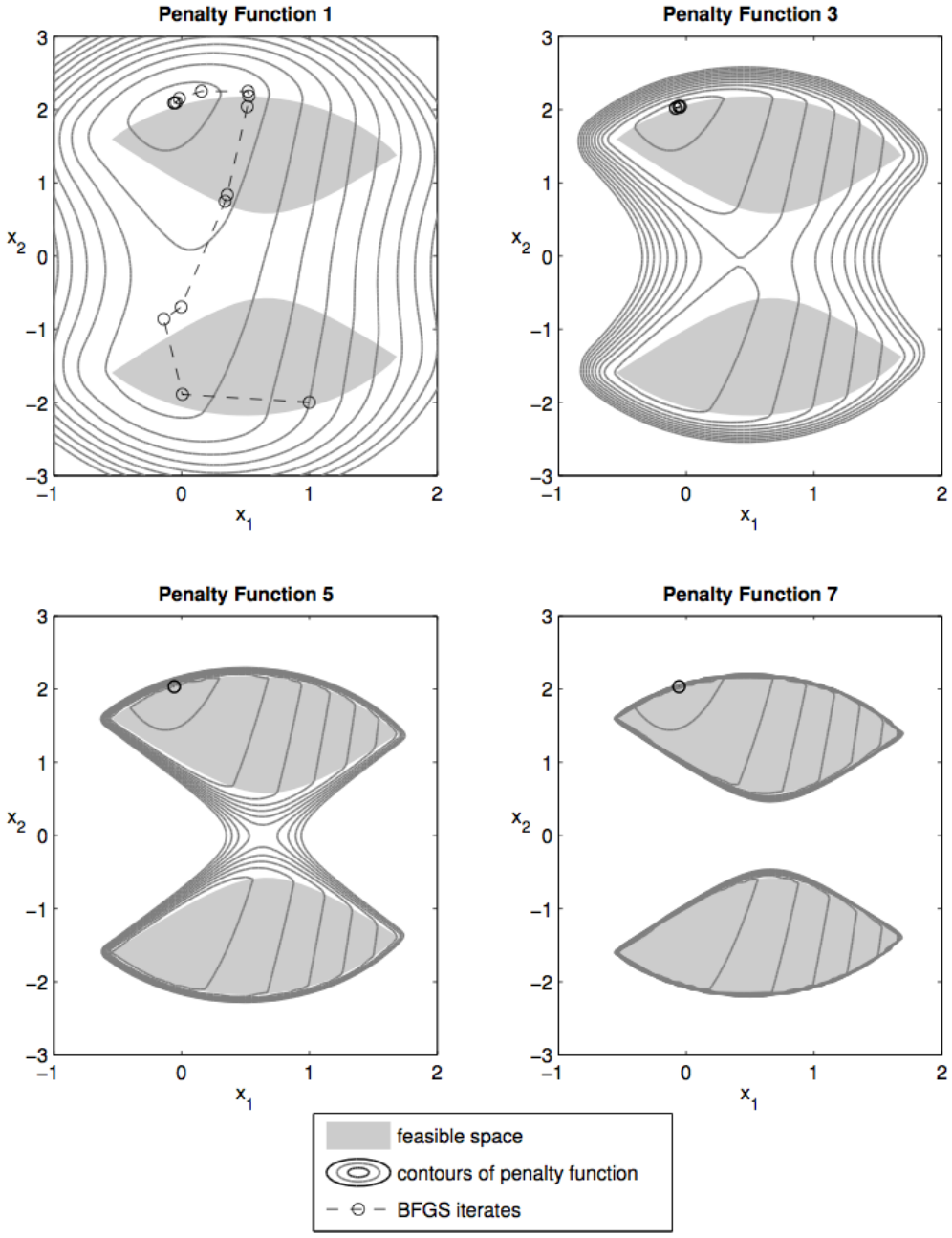


Figure 5.7: Solution of a constrained problem using a sequence of problems with quadratic penalty. Note that by starting with a low penalty parameter, the local optimum was avoided.

5.4 Sequential Quadratic Programming (SQP)

To understand the use of SQP in problems with inequality constraints, we begin by considering the equality-constrained problem,

$$\begin{aligned} \text{minimize} \quad & f(x) \\ \text{subject to} \quad & \hat{c}_j(x) = 0, \quad j = 1, \ldots, \hat{m} \end{aligned}$$

The idea of SQP is to model this problem at the current point x_k by a quadratic subproblem and to use the solution of this subproblem to find the new point x_{k+1}. SQP represents, in a way, the application of Newton's method to the KKT optimality conditions.

The Lagrangian function for this problem is $\mathcal{L}(x, \hat{\lambda}) = f(x) - \hat{\lambda}^T \hat{c}(x)$. We define the Jacobian of the constraints by

$$A(x)^T = \nabla \hat{c}(x)^T = [\nabla \hat{c}_1(x), \ldots, \nabla \hat{c}_{\hat{m}}(x)] \tag{5.64}$$

which is an $n \times m$ matrix and $g(x) \equiv \nabla f(x)$ is an n-vector as before. Note that A is generally not symmetric.

Applying the first order KKT conditions to this problem we obtain

$$\nabla \mathcal{L}(x, \hat{\lambda}) = 0 \Rightarrow \begin{bmatrix} g(x) - A(x)^T \hat{\lambda} \\ \hat{c}(x) \end{bmatrix} = 0 \tag{5.65}$$

This set of nonlinear equations can be solved using Newton's method,

$$\begin{bmatrix} W(x_k, \hat{\lambda}_k) & -A(x_k)^T \\ A(x_k) & 0 \end{bmatrix} \begin{bmatrix} p_k \\ p_{\hat{\lambda}} \end{bmatrix} = \begin{bmatrix} -g_k + A_k^T \hat{\lambda}_k \\ -\hat{c}_k \end{bmatrix} \tag{5.66}$$

where the Hessian of the Lagrangian is denoted by $W(x, \hat{\lambda}) = \nabla^2_{xx} \mathcal{L}(x, \hat{\lambda})$ and the Newton step from the current point is given by

$$\begin{bmatrix} x_{k+1} \\ \hat{\lambda}_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ \hat{\lambda}_k \end{bmatrix} + \begin{bmatrix} p_k \\ p_{\hat{\lambda}} \end{bmatrix}. \tag{5.67}$$

An alternative way of looking at this formulation of the SQP is to define the following quadratic problem at $(x_k, \hat{\lambda}_k)$

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2} p^T W_k p + g_k^T p \\ \text{subject to} \quad & A_k p + \hat{c}_k = 0 \end{aligned}$$

This problem has a unique solution that satisfies

$$\begin{aligned} W_k p + g_k - A_k^T \hat{\lambda}_k = 0 \\ A_k p + \hat{c}_k = 0 \end{aligned}$$

By writing this in matrix form, we see that p_k and $\hat{\lambda}_k$ can be identified as the solution of the Newton equations we derived previously.

$$\begin{bmatrix} W_k & -A_k^T \\ A_k & 0 \end{bmatrix} \begin{bmatrix} p_k \\ \hat{\lambda}_{k+1} \end{bmatrix} = \begin{bmatrix} -g_k \\ -\hat{c}_k. \end{bmatrix} \tag{5.68}$$

This problem is equivalent to (5.66), but the second set of variables, is now the actual vector of Lagrange multipliers $\hat{\lambda}_{k+1}$ instead of the Lagrange multiplier step, $p_{\hat{\lambda}}$.

5.4.1 Quasi-Newton Approximations

Any SQP method relies on a choice of W_k (an approximation of the Hessian of the Lagrangian) in the quadratic model. When W_k is exact, then the SQP becomes the Newton method applied to the optimality conditions.

One way to approximate the Hessian of the Lagrangian would be to use a quasi-Newton approximation, such as the BFGS update formula. We could define,

$$s_k = x_{k+1} - x_k, \qquad y_k = \nabla_x \mathcal{L}(x_{k+1}, \lambda_{k+1}) - \nabla_x \mathcal{L}(x_k, \lambda_{k+1}), \tag{5.69}$$

and then compute the new approximation B_{k+1} using the same formula used in the unconstrained case.

If $\nabla^2_{xx}\mathcal{L}$ is positive definite at the sequence of points x_k, the method will converge rapidly, just as in the unconstrained case. If, however, $\nabla^2_{xx}\mathcal{L}$ is not positive definite, then using the BFGS update may not work well.

To ensure that the update is always well-defined the *damped BFGS updating for SQP* was devised. Using this scheme, we set

$$r_k = \theta_k y_k + (1 - \theta_k) B_k s_k, \tag{5.70}$$

where the scalar θ_k is defined as

$$\theta_k = \begin{cases} 1 & \text{if} \quad s_k^T y_k \geq 0.2 s_k^T B_k s_k, \\ \frac{0.8 s_k^T B_k s_k}{s_k^T B_k s_k - s_k^T y_k} & \text{if} \quad s_k^T y_k < 0.2 s_k^T B_k s_k. \end{cases} \tag{5.71}$$

Then we can update B_{k+1} using,

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{r_k r_k^T}{s_k^T r_k}, \tag{5.72}$$

which is the standard BFGS update formula with y_k replaced by r_k. This guarantees that the Hessian approximation is positive definite.

Note that when $\theta_k = 0$ we have $B_{k+1} = B_k$, and that $\theta_k = 1$ yields an unmodified BFGS update. The modified method thus produces an interpolation between the current B_k and the one corresponding to BFGS. The choice of θ_k ensures that the new approximation stays close enough to the current approximation to guarantee positive definiteness.

In addition to using a different quasi-Newton update, SQP algorithms also need modifications to the line search criteria in order to ensure that the method converges from remote starting points. It is common to use a *merit function*, ϕ to control the size of the steps in the line search. The following is one of the possibilities for such a function:

$$\phi(x_k; \mu) = f(x) + \frac{1}{\mu} ||\hat{c}||_1 \tag{5.73}$$

The penalty parameter μ is positive and the L_1 norm of the equality constraints is

$$||\hat{c}||_1 = \sum_{j=1}^{\hat{m}} |\hat{c}_j|. \tag{5.74}$$

To determine the sequence of penalty parameters, the following strategy is often used

$$\mu_k = \begin{cases} \mu_{k-1} & \text{if} \quad \mu_{k-1}^{-1} \geq \gamma + \delta \\ (\gamma + 2\delta)^{-1} & \text{otherwise,} \end{cases} \tag{5.75}$$

where γ is set to $\max(\lambda_{k+1})$ and δ is a small tolerance that should be larger that the expected relative precision of your function evaluations.

The full procedure an SQP with line search is described in Algorithm 12, where D denotes the directional derivative in the p_k direction.

Algorithm 12 SQP algorithm

Input: Initial guess $(x_0, \hat{\lambda}_0)$, parameters $0 < \eta < 0.5$
Output: Optimum, x^*
  $k \leftarrow 0$
  Initialize the Hessian estimate, $B_0 \leftarrow I$
  repeat
    Compute p_k and $p_{\hat{\lambda}}$ by solving (5.66), with B_k in place of W_k
    Choose μ_k such that p_k is a descent direction for ϕ at x_k
    $\alpha_k \leftarrow 1$
    while $\phi(x_k + \alpha_k p_k, \mu_k) > \phi(x_k, \mu_k) + \eta \alpha_k D\left[\phi(x_k, p_k)\right]$ **do**
      $\alpha_k \leftarrow \tau_\alpha \alpha_k$ for some $0 < \tau_\alpha < 1$
    end while
    $x_{k+1} \leftarrow x_k + \alpha_k p_k$
    $\hat{\lambda}_{k+1} \leftarrow \hat{\lambda}_k + p_{\hat{\lambda}}$
    Evaluate f_{k+1}, g_{k+1}, c_{k+1} and A_{k+1}
    $s_k \leftarrow \alpha_k p_k$, $y_k \leftarrow \nabla_x \mathcal{L}(x_{k+1}, \hat{\lambda}_{k+1}) - \nabla_x \mathcal{L}(x_k, \hat{\lambda}_{k+1})$
    Obtain B_{k+1} by using a quasi-Newton update to B_k
    $k \leftarrow k + 1$
  until Convergence

5.4.2 Inequality Constraints

The SQP method can be extended to handle inequality constraints. Consider general nonlinear optimization problem

$$\begin{aligned} \text{minimize} \quad & f(x) \\ \text{subject to} \quad & \hat{c}_j(x) = 0, \quad j = 1, \ldots, \hat{m} \\ & c_k(x) \geq 0, \quad k = 1, \ldots, m \end{aligned}$$

To define the subproblem we now linearize both the inequality and equality constraints and obtain,

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2} p^T W_k p + g_k^T p \\ \text{subject to} \quad & \nabla \hat{c}_j(x)^T p + \hat{c}_j(x) = 0, \quad j = 1, \ldots, \hat{m} \\ & \nabla c_k(x)^T p + c_k(x) \geq 0, \quad k = 1, \ldots, m \end{aligned}$$

One of the most common type of strategy to solve this problem, the *active-set method*, is to consider only the active constraints at a given iteration and treat those as equality constraints. This is a significantly more difficult problem because we do not know *a priori* which inequality constraints are active at the solution. If we did, we could just solve the equality constrained problem considering only the active constraints.

The most commonly used active-set methods are feasible-point methods. These start with a feasible solution and never let the new point be infeasible.

Example 5.30. Constrained Minimization Using SQP

We consider the same inequality constrained problem 5.8 that we used to demonstrated the quadratic penalty approach. The sequence of subproblems determined by the SQP method is shown Fig. 5.8. The feasible regions are shown shaded, and the contours are those of the Lagrangian. Note that these contours change for each subproblem due to a change in the Lagrange multipliers. The intermediate Lagrangian functions are not necessarily convex, but the final one is, with an unconstrained minimum at the constrained optimum. In this case, the algorithm converges to the global optimum, but it could easily have converged to a the local one.

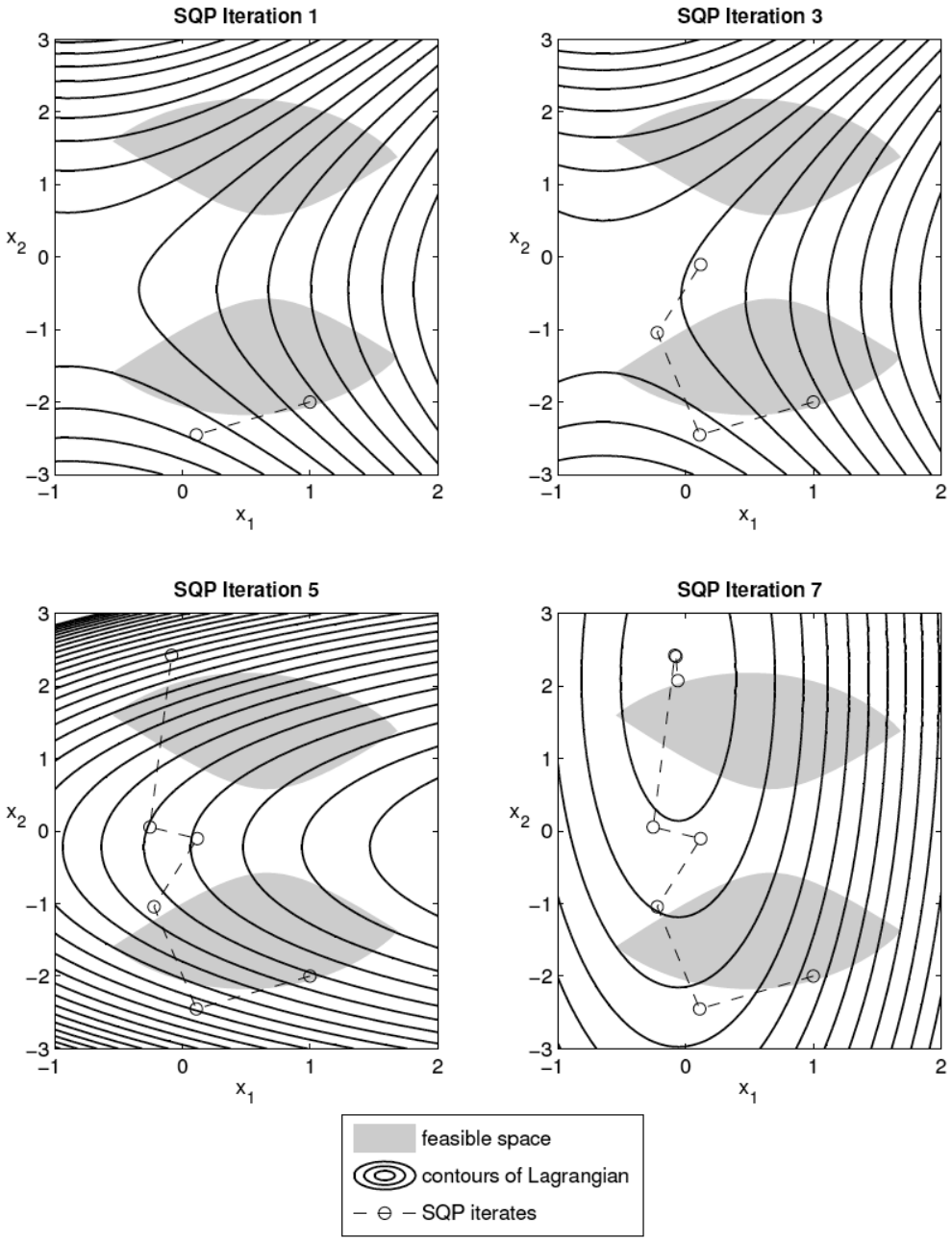


Figure 5.8: Solution of a constrained problem using SQP. Note that the Lagrangian is not always convex.

Bibliography

[1] Stephen G. Nash and Ariela Sofer. *Linear and Nonlinear Programming*, chapter 15. McGrawHill, 1996.

[2] Stephen G. Nash and Ariela Sofer. *Linear and Nonlinear Programming*, chapter 16. McGrawHill, 1996.

[3] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*, chapter 17. Springer-Verlag, 1999.

[4] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*, chapter 18. Springer-Verlag, 1999.

Chapter 6

Gradient-Free Optimization

6.1 Introduction

Using optimization in the solution of practical applications we often encounter one or more of the following challenges (see examples in Fig. 6.1):

- non-differentiable functions and/or constraints
- disconnected and/or non-convex feasible space
- discrete feasible space
- mixed variables (discrete, continuous, permutation)
- multiple local minima (multi-modal)
- multiple objectives

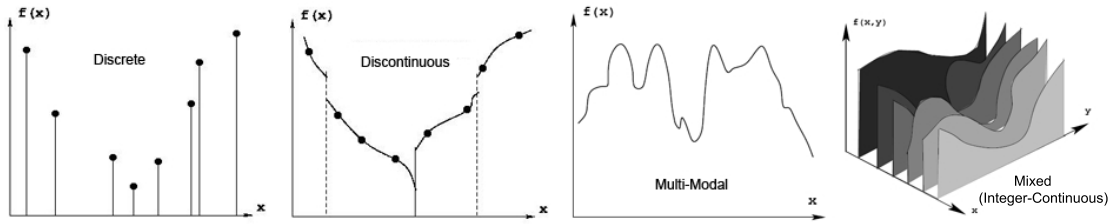


Figure 6.1: Graphs illustrating the various types of functions that are problematic for gradient-based optimization algorithms

Gradient-based optimizers are efficient at finding local minima for high-dimensional, nonlinearly-constrained, convex problems; however, most gradient-based optimizers have problems dealing with noisy and discontinuous functions, and they are not designed to handle highly multi-modal problems or discrete and mixed discrete-continuous design variables.

Consider, for example, the Griewank function (Fig. 6.2):

$$f(x) = \sum_{i=1}^{n} \left(\frac{x_i^2}{4000} \right) - \prod_{\mathrm{i}=1}^{\mathrm{n}} \cos\left(\frac{\mathrm{x_i}}{\sqrt{\mathrm{i}}} \right) + 1 \qquad -600 \leq \mathrm{x_i} \leq 600 \tag{6.1}$$

What are some approaches we can use to find *the best* solution for this example?

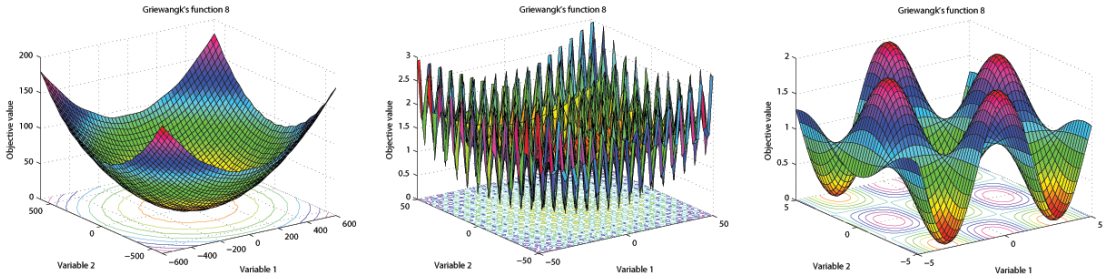


Figure 6.2: The Griewank function looks deceptively smooth when plotted in a large domain (left), but when you zoom in, you can see that the design space has multiple local minima (center) although the function is still smooth (right)

- Multiple point restarts of gradient (local) based optimizer
- Systematically search the design space
- Use gradient-free optimizers

Many gradient-free methods mimic mechanisms observed in nature or use heuristics. Unlike gradient-based methods in a convex search space, gradient-free methods are not necessarily guaranteed to find the true global optimal solutions, but they are able to find many good solutions (the mathematician's answer vs. the engineer's answer).

The key strength of gradient-free methods is their ability to solve problems that are difficult to solve using gradient-based methods. Furthermore, many of them are designed as global optimizers and thus are able to find multiple local optima while searching for the global optimum.

Various gradient-free methods have been developed. We are going to look at some of the most commonly used algorithms:

- Nelder–Mead Simplex (Nonlinear Simplex)
- Simulated Annealing
- Divided Rectangles Method
- Genetic Algorithms
- Particle Swarm Optimization

6.2 Nelder–Mead Simplex

The simplex method of Nelder and Mead performs a search in n-dimensional space using heuristic ideas. It is also known as the *nonlinear simplex* and is not to be confused with the linear simplex, with which it has nothing in common.

Its main strengths are that it requires no derivatives to be computed and that it does not require the objective function to be smooth. The weakness of this method is that it is not very efficient, particularly for problems with more than about 10 design variables; above this number of variables convergence becomes increasingly difficult.

A *simplex* is a structure in n-dimensional space formed by $n+1$ points that are not in the same plane. Hence a line segment is a 1-dimensional simplex, a triangle is a 2-dimensional simplex and a tetrahedron forms a simplex in 3-dimensional space. The simplex is also called a hypertetrahedron.

The Nelder–Mead algorithm starts with a simplex ($n+1$ sets of design variables x) and then modifies the simplex at each iteration using four simple operations. The sequence of operations to be performed is chosen based on the relative values of the objective function at each of the points.

The first step of the simplex algorithm is to find the $n+1$ points of the simplex given an initial guess x_0. This can be easily done by simply adding a step to each component of x_0 to generate n new points.

However, generating a simplex with equal length edges is preferable. Suppose the length of all sides is required to be c and that the initial guess, x_0 is the $(n+1)^{\text{th}}$ point. The remaining points of the simplex, $i = 1, \ldots, n$ can be computed by adding a vector to x_0 whose components are all b except for the i^{th} component which is set to a, where

$$b = \frac{c}{n\sqrt{2}}\left(\sqrt{n+1} - 1\right) \tag{6.2}$$

$$a = b + \frac{c}{\sqrt{2}}. \tag{6.3}$$

After generating the initial simplex, we have to evaluate the objective function at each of its vertices in order to identify three key points: the points associated with the highest ("worst") the second highest ("lousy") and the lowest ("best") values of the objective.

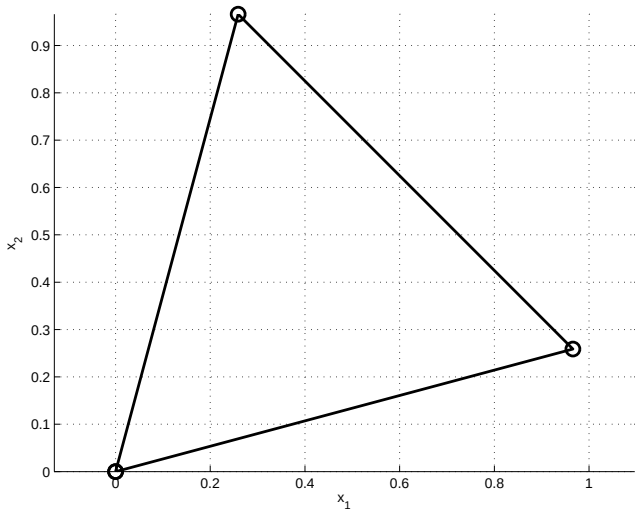


Figure 6.3: Starting simplex for $n = 2$, $x_0 = [0, 0]^T$ and $c = 1$. The two other vertices are $[a, b]$ and $[b, a]$.

The Nelder–Mead algorithm performs four main operations to the simplex: *reflection*, *expansion*, *outside contraction*, *inside contraction* and *shrinking* (Fig. 6.5). Each of these operations generates a new point (or points in the case of shrinking). The sequence of operations performed in one iteration depends on the value of the objective at the new point relative to the other key points.

Let x_w, x_l and x_b denote the worst, the second worst (or "lousy") and best points, respectively, among all $n+1$ points of the simplex. We first calculate the average of the n points that exclude the worst,

$$x_a = \frac{1}{n}\sum_{i=1, i\neq w}^{n+1} x_i. \tag{6.4}$$

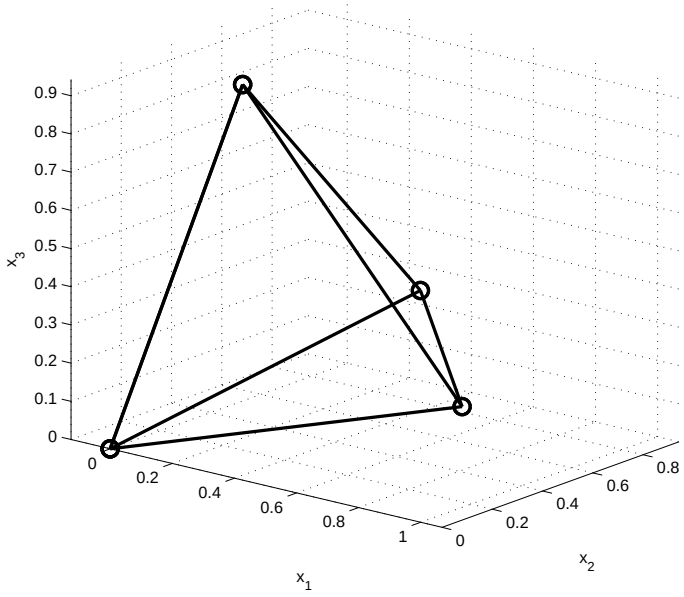


Figure 6.4: Starting simplex for $n = 3$, $x_0 = [0, 0, 0]^T$ and $c = 1$. The three other vertices are $[a, b, b]$, $[b, a, b]$ and $[b, b, a]$.

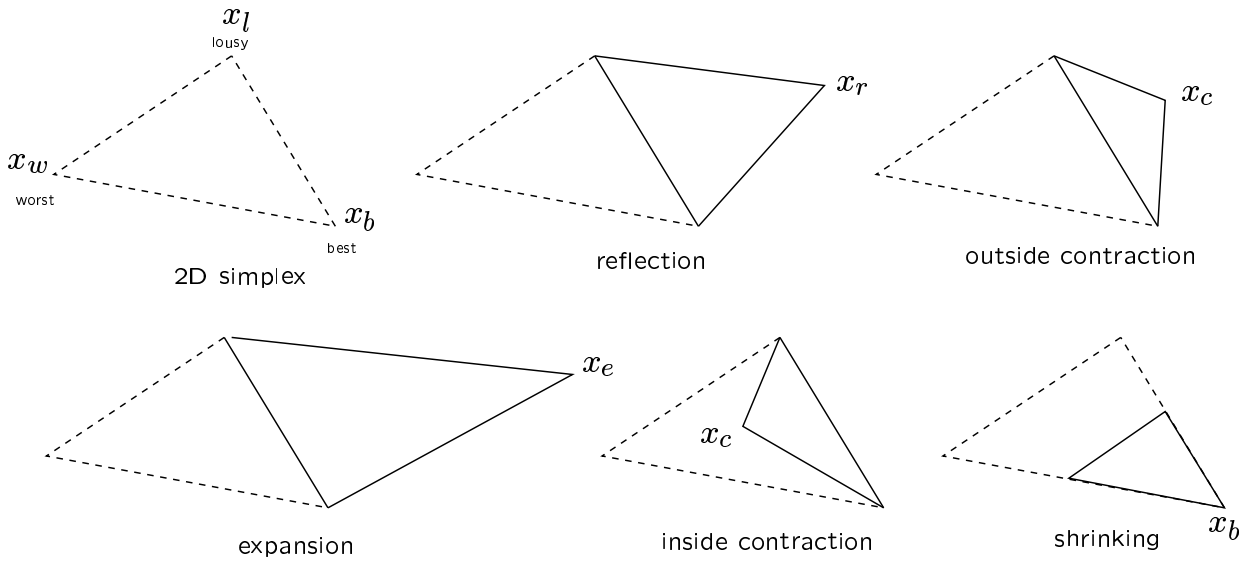


Figure 6.5: Operations performed on the simplex in Nelder–Mead's algorithm for $n = 2$.

After computing x_a, we know that the line from x_w to x_a is a descent direction. A new point is found on this line by *reflection*, given by

$$x_r = x_a + \alpha_r (x_a - x_w) \tag{6.5}$$

where $\alpha_r = 1$. If the value of the function at this reflected point is better than the best point, then the reflection has been especially successful and we step further in the same direction by performing an *expansion*,

$$x_e = x_r + \alpha_e (x_r - x_a), \tag{6.6}$$

where the expansion parameter α_e is usually set to 1. After the expansion, we accept the new point if the value of the objective function is better than the best point. Otherwise, we just accept the previous reflected point.

If, however the reflected point is worse than the worst point, we assume that a better point exists between x_w and x_a and perform an *inside contraction*

$$x_c = x_a - \beta (x_a - x_w), \tag{6.7}$$

where the contraction factor is usually set to $\beta = 0.5$. If the reflected point is not worse than the worst but is still worse than the lousy point, then an *outside contraction* is performed, that is,

$$x_o = x_a + \beta (x_a - x_w). \tag{6.8}$$

The remaining possibility for the reflected point is that it is not better than the best, but also not worse than the lousy point. In this case we simply accept the reflected point.

If either of the contraction operations fail (inside contraction fails if it is worse than the worst point, and outside contraction fails if it is worse than the reflected point), we resort to a *shrinking* operation. This operation retains the best point and shrinks the simplex, that is, for all point of the simplex except the best one, we compute a new position

$$x_i = x_b + \rho (x_i - x_b), \tag{6.9}$$

where the scaling parameter is usually set to $\rho = 0.5$.

The simplex operations are shown in Fig. 6.5 for the $n = 2$ case. A flow chart of the algorithm is shown in Fig. 6.6 and the algorithm itself in Algorithm 13.

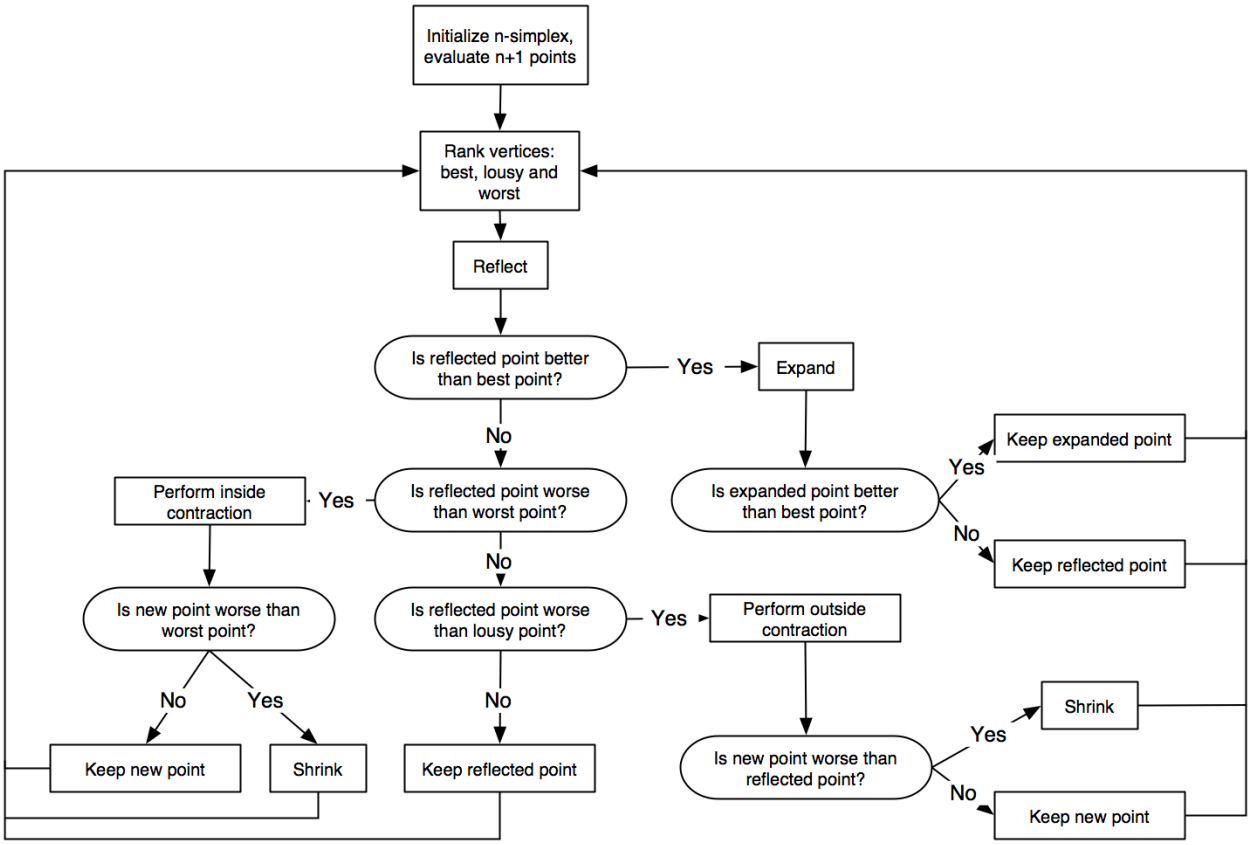


Figure 6.6: Flow chart for the Nelder–Mead algorithm. The best point is that with the lowest value of the objective function, the worst point is that with the highest value and the lousy point has the second highest value.

Algorithm 13 Nelder–Mead Algorithm

Input: Initial guess, x_0
Output: Optimum, x^*
  $k \leftarrow 0$
  Create a simplex with edge length c
  repeat
    Identify the highest (x_w: worst), second highest (x_l, lousy) and lowest (x_b: best) value points with function values f_w, f_l, and f_b, respectively
    Evaluate x_a, the average of the point in simplex excluding x_w
    Perform *reflection* to obtain $x_r = x_a + \alpha_r (x_a - x_w)$, evaluate f_r
    if $f_r < f_b$ **then**
      Perform *expansion* to obtain $x_e = x_r + \alpha_e (x_r - x_a)$, evaluate f_e.
      if $f_e < f_b$ **then**
        $x_w \leftarrow x_e$, $f_w \leftarrow f_e$ (accept expansion)
      else
        $x_w \leftarrow x_r$, $f_w \leftarrow f_r$ (accept reflection)
      end if
    else if $f_r \leq f_l$ **then**
      $x_w \leftarrow x_r$, $f_w \leftarrow f_r$ (accept reflected point)
    else
      if $f_r > f_w$ **then**
        Perform an *inside contraction* to obtain $x_c = x_a - \beta (x_a - x_w)$, and evaluate f_c
        if $f_c < f_w$ **then**
          $x_w \leftarrow x_c$ (accept contraction)
        else
          Shrink the simplex
        end if
      else
        Perform an *outside contraction* to obtain $x_o = x_a + \beta (x_a - x_w)$, and evaluate f_o
        if $f_o \leq f_r$ **then**
          $x_w \leftarrow x_o$ (accept contraction)
        else
          Shrink the simplex
        end if
      end if
    end if
    $k \leftarrow k + 1$
  until $(f_w - f_b) < (\varepsilon_1 + \varepsilon_2 |f_b|)$

The convergence criterion can also be that the size of simplex, i.e.,

$$s = \sum_{i=1}^{n} |x_i - x_{n+1}| \tag{6.10}$$

must be less than a certain tolerance. Another measure of convergence that can be used is the standard deviation,

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n+1} \left(f_i - \bar{f}\right)^2}{n+1}}$$

where $\bar{f}$ is the mean of the $n+1$ function values.

Variations of the simplex algorithm described above have been tried, with interesting results. For example, note that if $f_e < f_b$ but f_r is even better, that is $f_r < f_e$, the algorithm still accepts the expanded point x_e. Now, it is standard practice to accept the best of f_r and f_e.

Note that the methodology (like most direct-search methods) cannot directly handle constraints. One approach to handle constraints is to use a penalty method to form a an unconstrained problem. In this case, the penalty need not be differentiable, so a linear penalty method would suffice.

Example 6.31. Minimization of the Rosenbrock Function Using Nelder–Meade

Fig. 6.7 shows the sequence of simplices that results when minimizing the Rosenbrock function. The initial simplex on the upper left is equilateral. The first iteration is an inside contraction, followed by a reflection, another inside contraction and then an expansion. The simplices then reduce dramatically in size and follow the Rosenbrock valley, slowly converging to the minimum.

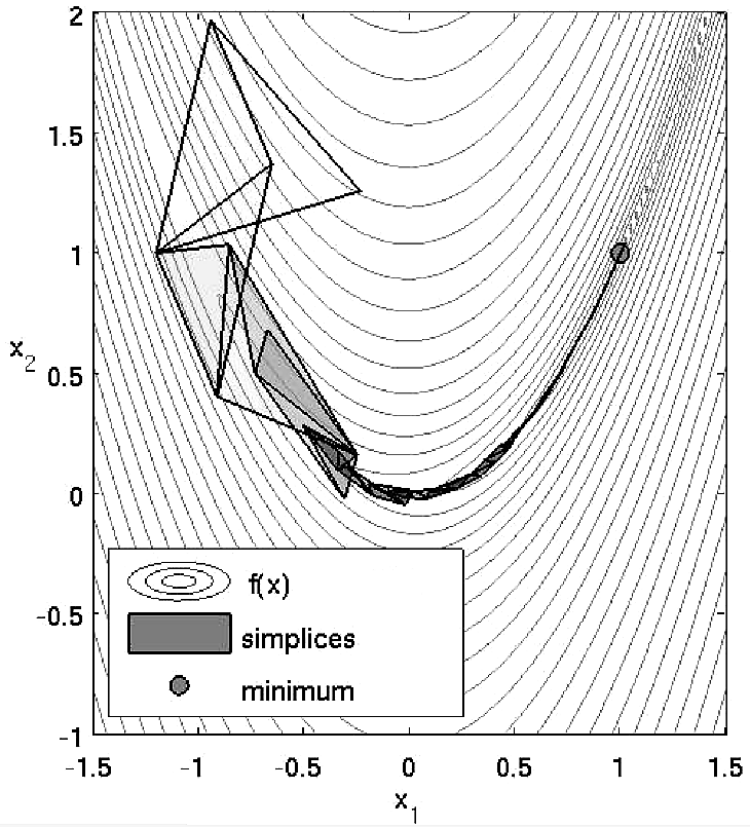


Figure 6.7: Sequence of simplices that minimize the Rosenbrock function

6.3 DIvided RECTangles (DIRECT) Method

As we saw before, one of the strategies to do global optimization is to perform a systematic search of the design space. The DIRECT method uses a hyperdimensional adaptive meshing scheme to search all the design space to find the optimum. The overall idea behind DIRECT is as follows.

1. The algorithm begins by scaling the design box to a n-dimensional unit hypercube and evaluating the objective function at the center point of the hypercube

2. The method then divides the potentially optimal hyper-rectangles by sampling the longest coordinate directions of the hyper-rectangle and trisecting based on the directions with the smallest function value until the global minimum is found

3. Sampling of the maximum length directions prevents boxes from becoming overly skewed and trisecting in the direction of the best function value allows the biggest rectangles contain the best function value. This strategy increases the attractiveness of searching near points with good function values

4. Iterating the above procedure allow to identify and zoom into the most promising design space regions

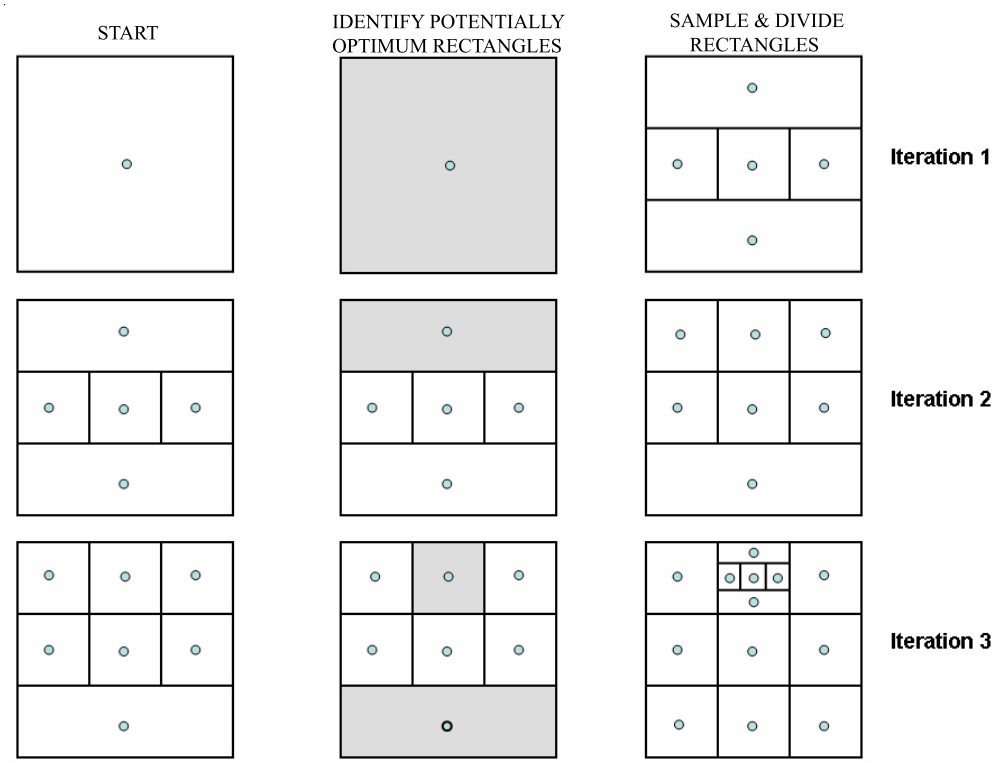


Figure 6.8: Example of DIRECT Iterations

To identify the *potentially optimal rectangles* we do the following. Assuming that the unit hypercube with center c_i is divided into m hyper-rectangles, a hyper-rectangle j is said to be potentially optimal if there exists rate-of-change constant $\bar{K} > 0$ such that

$$\begin{aligned} f(c_j) - \bar{K} d_j &\leq f(c_i) - \bar{K} d_i \quad \text{for all} \quad i = 1, ..., m \\ f(c_j) - \bar{K} d_j &\leq f_{\min} - \varepsilon |f_{min}|, \end{aligned} \tag{6.11}$$

where d is the distance between c and the vertices of the hyper-rectangle, $f_{\min}$ is the best current value of the objective function, and ε is positive parameter used so that $f(c_j)$ exceeds the current best solution by a non-trivial amount.

To visualize these equations, we plot the f versus the d for a given group of points in Fig. 6.9. The line connecting the points with the lowest f for a given d (or greatest d for a given f) represent the points with the most potential. The first equation forces the selection of the rectangles on

this line, and the second equation insists that the obtained function value exceeds the current best function value by an amount that is not insignificant. This prevents the algorithm from becoming too local, wasting precious function evaluations in search of smaller function improvements. The parameter ε balances the search between local and global. A typical value is $\varepsilon = 10^{-4}$, and its the range is usually such that $10^{-2} \leq \varepsilon \leq 10^{-7}$.

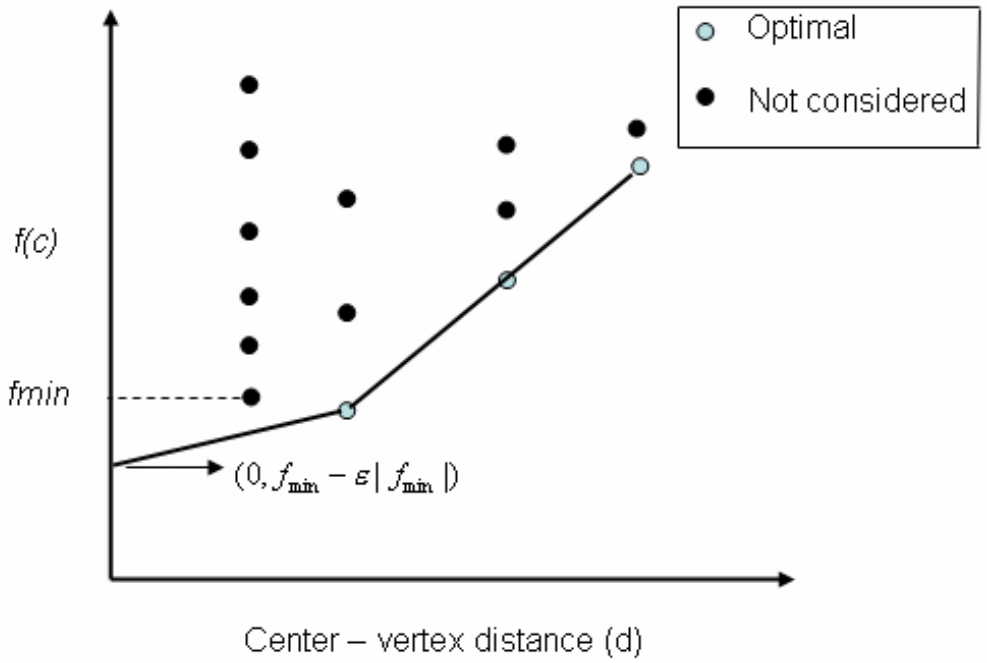


Figure 6.9: Identification of potentially optimal rectangles

Algorithm 14 DIRECT Algorithm

Input: Initial guess, x_0
Output: Optimum, x^*
  $k \leftarrow 0$
  repeat
    Normalize the search space to be the unit hypercube. Let c_1 be the center point of this hypercube and evaluate $f(c_1)$.
    Identify the set S of potentially optimal rectangles/cubes, that is all those rectangles defining the bottom of the convex hull of a scatter plot of rectangle diameter versus $f(c_i)$ for all rectangle centers c_i
    for all Rectangles $r \in S$ **do**
      Identify the set I of dimensions with the maximum side length
      Set δ equal one third of this maximum side length
      for all $i \in I$ **do**
        Evaluate the rectangle/cube at the point $c_r \pm \delta e_i$ for all $i \in I$, where c_r is the center of the rectangle r, and e_i is the i^{th} unit vector
      end for
      Divide the rectangle r into thirds along the dimensions in I, starting with the dimension with the lowest value of $f(c \pm \delta e_i)$ and continuing to the dimension with the highest $f(c \pm \delta e_i)$.
    end for
  until Converged

Example 6.32. Minimization of the Rosenbrock Function Using DIvided RECTangles Method

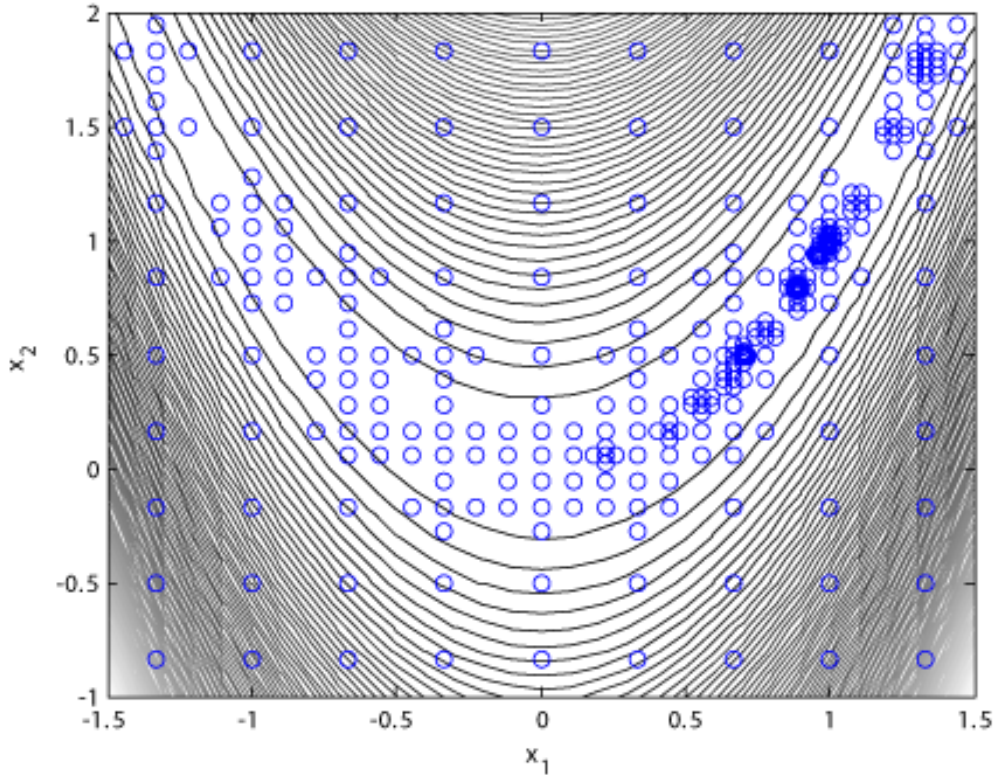


Figure 6.10: Minimization of the Rosenbrock function using DIRECT

6.4 Genetic Algorithms

Genetic algorithms for optimization were inspired by the process of natural evolution of organisms. This type of algorithm was first developed by John Holland in the mid 1960's. Holland was motivated by a desire to better understand the evolution of life by simulating it in a computer and the use of this process in optimization.

Genetic algorithms are based on three essential components:

- Survival of the fittest (Selection)
- Reproduction processes where genetic traits are propagated (Crossover)
- Variation (Mutation)

We will use the term "genetic algorithms" generically, which is the most common term used when expressing a large family of optimization approaches that use the three components above. Depending on the approach they have different names, for example: genetic algorithms, evolutionary computation, genetic programming, evolutionary programming, evolutionary strategies.

We will start by posing the unconstrained optimization problem with design variable bounds,

$$\begin{aligned} &\text{minimize} && f(x) \\ &\text{subject to} && x_l \leq x \leq x_u \end{aligned}$$

where x_l and x_u are the vectors of lower and upper bounds on x, respectively.

In the context of genetic algorithms we will call each design variable vector x a *population member*. The value of the objective function, $f(x)$ is termed the *fitness*.

Genetic algorithms are radically different from the gradient based methods we have covered so far. Instead of looking at *one* point at a time and stepping to a new point for each iteration, a whole *population* of solutions is iterated towards the optimum at the same time. Using a population lets us explore multiple "buckets" (local minima) simultaneously, increasing the likelihood of finding the global optimum.

The main advantages of genetic algorithms are:

- The basic algorithm can use a coding of the parameter set instead of the parameters themselves; consequently, the algorithm can handle mixed continuous, integer and discrete design variables.
- The population can cover a large range of the design space and is less likely than gradient based methods to "get stuck" in local minima.
- As with other gradient-free methods like Nelder–Mead's algorithm, it can handle noisy objective functions.
- The implementation is straightforward and easily parallelized.
- They can easily be used for *multiobjective optimization*.

There is "no free lunch," of course, and these methods have some disadvantages. The main one is that genetic algorithms are computationally expensive when compared to gradient-based methods, especially for problems with a large number of design variables. Although genetic algorithms are much better than completely random methods, they are still "brute force" methods that require a large number of function evaluations.

However, there are cases where it is difficult to make gradient-based methods work or where they do not work at all. In some of these problems genetic algorithms work very well with little effort.

Single-Objective Optimization

The procedure of a genetic algorithm can be described as follows:

1. **Initialize a Population**

   Each member of the population represents a design point, x and has a value of the objective (fitness), and information about its constraint violations associated with it.

2. **Determine Mating Pool**

   Each population member is paired for reproduction using various methods.

3. **Generate Offspring**

   To generate offspring we need a scheme for the crossover operation. There are various schemes that one can use. When the design variables are continuous, for example, one offspring can be found by interpolating between the two parents and the other one can be extrapolated in the direction of the fitter parent.

4. **Mutation**

   Add some randomness in the offspring's variables to maintain diversity.

5. **Compute Offspring's Fitness**

   Evaluate the value of the objective function and constraint violations for each offspring.

6. **Identify the Best Member**

   Convergence is difficult to determine because the best solution so far may be maintained for many generations. As a rule of thumb, if the best solution among the current population hasn't changed (much) for about 10 generations, it can be assumed as the optimum for the problem.

7. **Return to Step 2**

Note that since GAs are probabilistic methods (due to the random initial population and mutation), it is crucial to run the problem multiple times when studying its characteristics.

Multi-Objective Optimization

What if we want to investigate the trade-off between two (or more) conflicting objectives? In the design of a supersonic aircraft, for example, we might want to simultaneously minimize aerodynamic drag and sonic boom and we do not know what the trade-off is. How much would the drag increase for a given reduction in sonic boom?

In this situation there is no one "best design". There is a set (a population) of designs that are the best possible for that combination of the two objectives. In other words, for these optimal solutions, the only way to improve one objective is to worsen the other. This is a natural application for genetic algorithms. We already have to evaluate a whole population, so we can use this to our advantage. Gradient-based approaches can perform multi-objective optimization through a

composite weighted functions (e.g., $J = D + kW$), or by making of the objectives a constraint whose bound is repeatedly changed.

The concept of *dominance* is the key to the use of GA's in multi-objective optimization. As an example, assume we have a population of 3 members, A, B and C, and that we want to minimize two objective functions, f_1 and f_2 (Section 6.4).

Member	f_1	f_2
A	10	12
B	8	13
C	9	14

Comparing members A and B, we can see that A has a higher (worse) f_1 than B, but has a lower (better) f_2. Hence we cannot determine whether A is better than B or vice versa. On the other hand, B is clearly a fitter member than C since both of B's objectives are lower. We say that B *dominates* C. Comparing A and C, once again we are unable to say that one is better than the other.

In summary:

- A is non-dominated by either B or C
- B is non-dominated by either A or C
- C is dominated by B but not by A

The *rank* of a member is the number of members that dominate it plus one. In this case the ranks of the three members are:

$$\text{rank(A)} = 1$$
$$\text{rank(B)} = 1$$
$$\text{rank(C)} = 2$$

In multi-objective optimization the rank is crucial in determining which population member are the fittest. A solution of rank one is said to be *Pareto optimal* and the set of rank one points for a given generation is called the *Pareto set*. As the number of generations increases, and the fitness of the population improves, the size of the Pareto set grows. In the case above, the Pareto set includes A and B. The graphical representation of a Pareto set is called a *Pareto front*.

The procedure of a two-objective genetic algorithm is similar to the single-objective one, with the following modifications.

1. **Initialize a Population**

   Compute all objectives and rank the population according to the definition of dominance

2. **Determine Mating Pool**

   Fitness is now based on rank rather than the objective function.

3. **Generate Offspring**

4. **Mutation**

5. **Compute Offspring's Fitness**

   Evaluate their objectives and rank offspring population.

6. **Rank New Population**

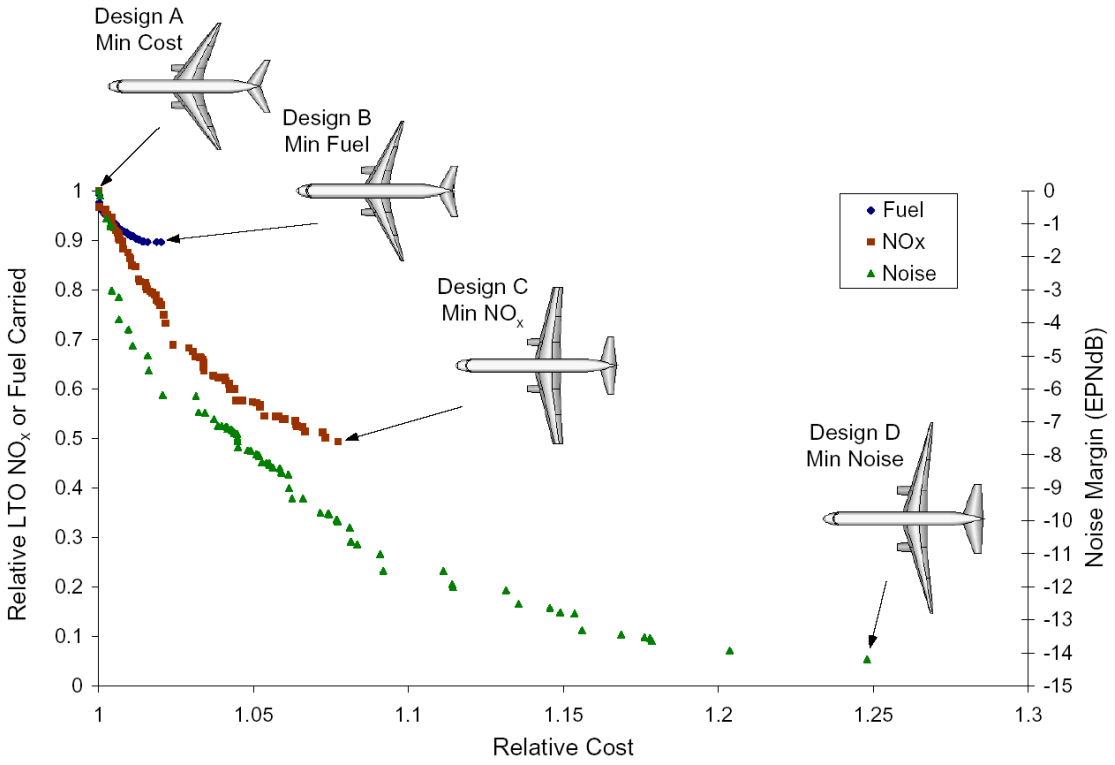
   Display rank one population, that is the Pareto set.

7. **Repeat From Step 2**

   We iterate until the Pareto front is no longer "moving".

One of the problems with this method is that there is no mechanism "pushing" the Pareto front to a better one. If one new member dominates every other member, the rank of the other members grows by one and the process continues. Some work has been invested in applying gradients to the front, with mixed results.

Example 6.33. Pareto Front in Aircraft Design [1]



6.4.1 Binary-coded Genetic Algorithms

The original genetic algorithms were based on binary encoding because they more naturally mimic genetics, namely chromosome-like encoding. Binary-coded GA's are still widely used, and are especially useful with discrete or mixed-integer problems. In a binary encoding we represent each variable as a binary number with m bits. If we want to represent a real-valued variable, we have to divide the feasible interval of x_i into $2^m - 1$ intervals. Then each possibility for x_i can be represented by any combination of m bits. For $m = 5$, for example, the number of intervals would be 31 and a possible representation for x_i would be 10101, which can be decoded as

$$x_i = x_l + s_i \left(1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0\right) = x_l + 21 s_i, \tag{6.12}$$

where s_i is the size of interval for x_i given by

$$s_i = \frac{x_{u_i} - x_{l_i}}{31}. \tag{6.13}$$

Often the encoding for each variable are combined into one large string, like a chromosome composed of many alleles. The binary-encodings must, of course, be decoded before use in function evaluations.

Creation of the Initial Population

As a rule of thumb, the population size should be of 15 to 20 times the number of design variables, but in general you will need to experiment with different population sizes.

Using bit encoding, each bit is assigned a 50% chance of being either 1 or 0. One way of doing this is to generate a random number $0 \leq r \leq 1$ and setting the bit to 0 if $r \leq 0.5$ and 1 if $r > 0.5$.

Selection: Determining the Mating Pool

In this phase bad solutions are eliminated from the population, and multiple copies of good solutions are added to the mating pool. The mating pool will thus have a higher average fitness than the average fitness of the population. Many different methods exist for selection, two simple ones are discussed below.

In one version of *tournament selection* two designs are selected at random, compared, and the better design is placed in the mating pool. This procedure is repeated for another pair of designs, until every design has participated in two separate tournaments. The best design will win both of its tournaments, and consequently will have two copies in the mating pool. The worst design will lose both of its tournaments, and consequently will be eliminated from the population. In the end, all designs will appear zero, one, or two times in the mating pool.

An alternative approach is called the *roulette wheel selection*. The basic idea is that members with a higher fitness should have a proportionally higher chance of being selection (think of a larger wedge on a roulette wheel).

It is convenient to scale the function values such that they all become positive, we would also like the lowest fitness to be larger than 0. Consider the highest (best) and the lowest (worst) values in the population, f_h and f_l respectively,. The function values can be converted to a positive quantity by adding,

$$C = 0.1f_h - 1.1f_l \tag{6.14}$$

to each function value. Thus the new highest value will be $1.1(f_h - f_l)$ and the new lowest value $0.1(f_h - f_l)$. The values are then adjusted as,

$$f_i' = f_i + C \tag{6.15}$$

We can then take the normalized cumulative sum of the scaled fitness values to compute N intervals where N is the number of members in the population

$$s_j = \frac{\sum_{i=1}^{j} f_i'}{\sum_{i=1}^{N} f_i'} \tag{6.16}$$

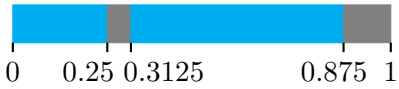
Note that the roulette wheel approach assigns a higher probability to *higher fitness* values. This means that if we want to perform minimization we need to make a minor change. Some options include setting $f = -f$ before Eq. (6.14) to turn it into a minimization, or we could invert $f' = 1/f'$ before Eq. (6.16) so that smaller numbers carry a larger final weight.

A mating pool of N members is created by turning the roulette wheel N times. A random number $0 \leq r \leq 1$ is generated at each turn. The j^{th} member is copied to the mating pool if

$$r \leq s_j \tag{6.17}$$

This ensures that the probability of a member being selected for reproduction is proportional to its scaled fitness value.

As an example, assume that $f' = [20, 5, 45, 10]$. Then $s = [0.25, 0.3125, 0.875, 1]$, which divides the "wheel" into four segments shown graphically as:



Crossover Operation (Generating Offspring)

Various crossover strategies are possible in genetic algorithms. *Single-point crossover* usually involves generating a random integer $1 \leq k \leq m - 1$ that defines the *crossover point*. For one of the offspring, the first k bits are taken from say parent 1 and the remaining bits from parent 2. For the second offspring, the first k bits are taken from parent 2 and the remaining ones from parent 1.

Before Crossover	After Crossover
11 111	11 000
00 000	00 111

Mutation

Mutation is a random operation performed to change the genetic information. Mutation is needed because even though reproduction and crossover effectively recombine existing information, occasionally some useful genetic information might be lost. The mutation operation protects against such irrecoverable loss. It also introduces additional diversity into the population.

When using bit representation, every bit is assigned a small permutation probability, say $p = 0.005 \sim 0.1$. This is done by generating a random number $0 \leq r \leq 1$ for each bit, which is changed if $r < p$.

Before Mutation	After Mutation
11111	11010

6.4.2 Real-Coded Genetic Algorithms

Real-coded GAs offer several advantages:

- can represent arbitrary precision (up to machine precision) rather than being limited by the initial choice in string length for binary GAs
- avoids "Hamming cliffs" where a large binary change is required between adjacent real numbers (e.g., 0111 to 1000)

- avoids binary coding/decoding

The disadvantages are that crossover and mutation are less natural and convenient. Integer or discrete values can be handled in a real-coded GA, but requires finding a nearest value. Changes in the operations for real-coded GAs are described in the following sections.

Creation of the Initial Population

Each member is often chosen at random within some initial bounds. For each design variable x_i, with bounds such that $x_{li} \leq x_i \leq x_{ui}$, we could use,

$$x_i = x_{li} + r(x_{ui} - x_{li}) \tag{6.18}$$

where r is a random number such that $0 \leq r \leq 1$.

Selection: Determining the Mating Pool

No change needed.

Crossover Operation (Generating Offspring)

Many crossover options exist. A common method is *linear crossover*:

$$x_c = w_1 x_{p1} + w_2 x_{p2} \tag{6.19}$$

where each x is a vector. With linear crossover 2 or more children solutions are generated and the two best offspring replace the parents. Sometimes the evaluation of two best includes the paresnts. A common weighting is equal weighting ($w_1 = w_2 = 0.5$). Additional offspring may bias towards the better parent ($w_1 = 2, w_2 = 1$), or may try multiple evaluations with biased weighting for each parent separately ($w_1 = 1.5, w_2 = -0.5$; $w_1 = 0.5, w_2 = 1.5$).

Another option is a simple crossover like the binary case where a random integer is generated to split the vectors. For example with a split after the first index:

$$\begin{aligned} x_{p1} &= [x_1, x_2, x_3, x_4] \\ x_{p2} &= [x_5, x_6, x_7, x_8] \\ \Rightarrow x_{c1} &= [x_1, x_6, x_7, x_8] \\ x_{c2} &= [x_5, x_2, x_3, x_4] \end{aligned} \tag{6.20}$$

This simple crossover does not generate as much diversity as the binary case does and relies more heavily on effective mutation.

Mutation

Many mutation methods rely on random variations around an existing member such as a uniform random operator:

$$x_{new\,i} = x_i + (r_i - 0.5)\Delta_i \tag{6.21}$$

where r_i is a random number between 0 and 1, and Δ_i is a pre-selected maximum perturbation in the i^{th} direction.

Many non-uniform methods exist as well, for example, using a Gaussian distribution

$$x_{new\,i} = x_i + \mathcal{N}(0, \sigma_i) \tag{6.22}$$

where σ_i is a pre-selected standard deviation and random samples are drawn from the normal distribution. During the mutation operations, bound checking is necessary to ensure the mutations stay within the upper and lower limits.

6.4.3 Constraint Handling

Various approaches exist for handling constraints. Like the Nelder-Mead method we can use a penalty method (e.g., Augmented Lagrangian, linear penalty, etc.). However, there are additional options for GAs. In the tournament selection we can use other selection criteria that do not depend on penalty parameters. One such approach for choosing the best selection amongst two competitors is:

1. prefer a feasible solution
2. among two feasible solutions choose the one with a better objective
3. among two infeasible solutions choose the one with a smaller constraint violation

6.4.4 Why do genetic algorithms work?

A fundamental question which is still being researched is how the three main operations (Selection,Crossover and Mutation) are able to find better solutions. Two main mechanism allow the algorithm to progress towards better solutions:

Selection + Mutation = Improvement: Mutation makes local changes while selection accepts better changes, this can be seen as a resilient and general form of reducing the objective function.

Selection + Crossover = Innovation: When the information of the best population members is exchanged, there is a greater chance a new better combination will emerge.

Example 6.34. Jet Engine Design at General Electric [12]

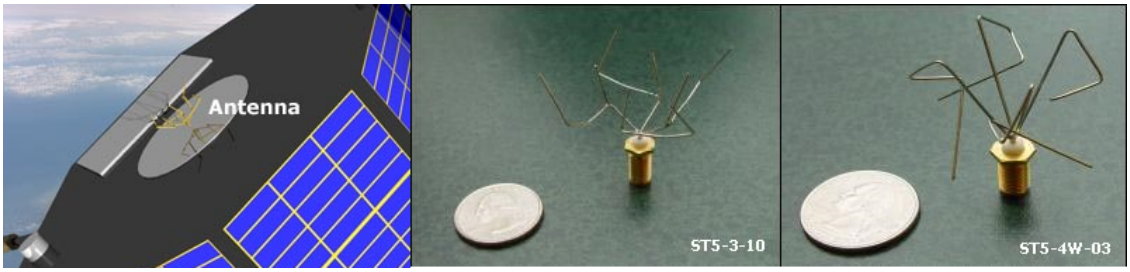
- Genetic algorithm combined with expert system
- Find the most efficient shape for the fan blades in the GE90 jet engines
- 100 design variables
- Found 2% increase in efficiency as compared to previous engines
- Allowed the elimination of one stage of the engine's compressor reducing engine weight and manufacturing costs without any sacrifice in performance

Example 6.35. ST5 Antenna

The antenna for the ST5 satellite system presented a challenging design problem, requiring both a wide beam width for a circularly-polarized wave and a wide bandwidth.

The GA found an antenna configuration (ST5-3-10) that was slightly more difficult to manufacture, but:

- Used less power
- Removed two steps in design and fabrication
- Had more uniform coverage and wider range of operational elevation angle relative to the ground changes
- Took 3 person-months to design and fabricate the first prototype as compared to 5 person-months for the conventionally designed antenna.



6.5 Particle Swarm Optimization

PSO is a stochastic, population-based computer algorithm developed in 1995 by James Kennedy (social-psychologist) and Russell Eberhart (electrical engineer) it applies the concept of swarm intelligence (SI) to problem solving [5].

What is Swarm Intelligence?

SI is the property of a system whereby the collective behaviors of (unsophisticated) agents interacting locally with their environment cause coherent functional global patterns to emerge (e.g., self-organization, emergent behavior).

Basic PSO Formulation

- Each agent (or particle) represents a design point and moves in n-dimensional space looking for the best solution.
- Each agent adjusts its movement according to the effects of cognitivism (self experience) and sociocognition (social interaction). In other words, each agent remembers the location where it found its best result so far, and it exchanges information with the swarm about the location where the swarm has found the best result so far.

At each time step a particle adjusts its velocity with an inertial component (a continuation in its previous direction), a weighted random component toward its personal best location, and a weighted random component toward the swarm's best location.

$$v_{k+1}^i = wv_k^i + c_1 r_1 \frac{\left(p_k^i - x_k^i\right)}{\Delta t} + c_2 r_2 \frac{\left(p_k^g - x_k^i\right)}{\Delta t} \tag{6.23}$$

With a new velocity, particle i simply updates its position as:

$$x_{k+1}^i = x_k^i + v_{k+1}^i \Delta t \tag{6.24}$$

The "time" dependence is artificial, and usually we eliminate it by multiplying Eq. (6.23) by Δt

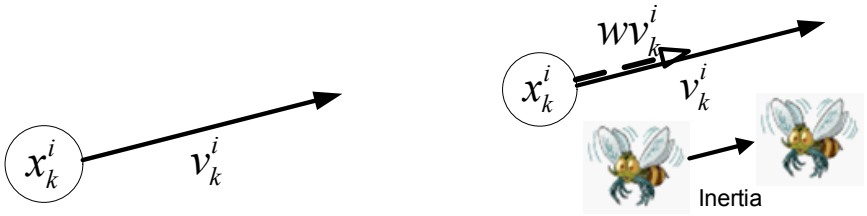
$$\bar{v}_{k+1}^i = \bar{w}\bar{v}_k^i + c_1 r_1 \left(p_k^i - x_k^i\right) + c_2 r_2 \left(p_k^g - x_k^i\right) \tag{6.25}$$

and the position update is then given by (for simplicity, all bars are typically dropped)

$$x_{k+1}^i = x_k^i + \bar{v}_{k+1}^i \tag{6.26}$$

- p_k^i is particle i's best position so far, p_k^g is the swarm's best particle position so far.
- c_1 is the cognitive parameter in the interval [0, 2] (confidence in itself), c_2 is the social parameter in the interval [0, 2] (confidence in the swarm). Both coefficients are typically close to 2. The cognitive parameter causes the particle to move toward the best region it has found. The social parameter causes the particle to to move toward the best region that the swarm has found.
- $\bar{w}$ is the inertia parameter in the interval [0, 1.2] Typical inertial coefficients are between 0.8 and 1.2. A lower value of $\bar{w}$ dampens the particle's inertia and tends toward faster convergence to an minimum. A higher value of $\bar{w}$ accelerates the particle's inertia and tends toward increased exploration to potentially help discover multiple minima.
- r_1 and r_2 are random numbers in the interval $[0, 1]$.

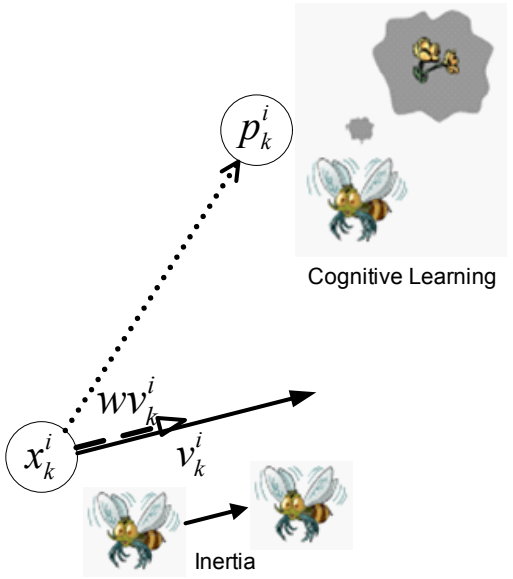
Figures 6.11 to 6.13 describe how the particle swarm is updated.



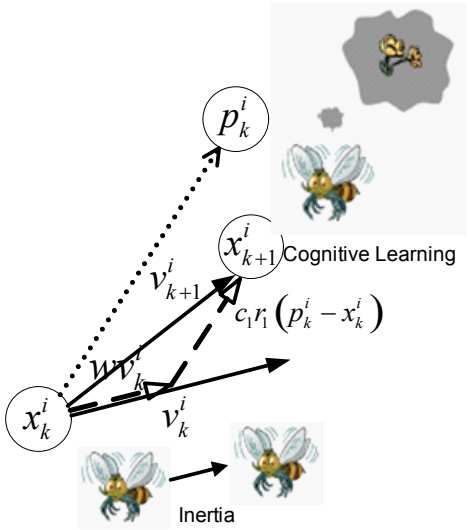
(a) The current position and velocity vector.

(b) The inertial component consist of a weighted speed in the current direction.

Figure 6.11: Inertial Component.

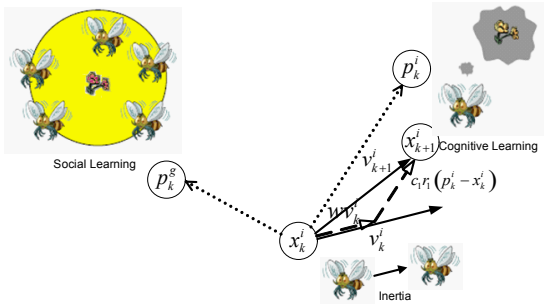


(a) The point p_k^i is the best point this agent has found so far.

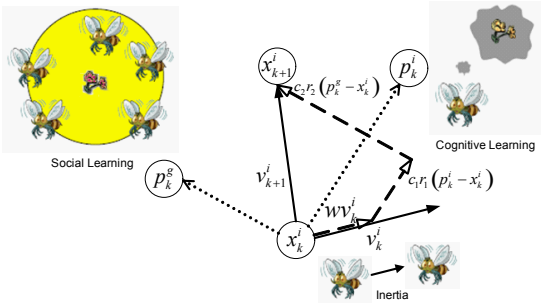


(b) The cognitive component consists of a (random) weighted velocity towards p_k^i.

Figure 6.12: Cognitive Component.



(a) The point p_k^g is the best point the swarm has found so far.



(b) The social component consists of a (random) weighted velocity towards p_k^g. These three components (inertial, cognitive, social) comprise the next direction vector for the agent.

Figure 6.13: Social Component.

PSO Algorithm

1. Initialize a set of particles positions x_o^i and velocities v_o^i randomly distributed throughout the design space bounded by specified limits.
2. Evaluate the objective function values $f\left(x_k^i\right)$ using the design space positions x_k^i.
3. Update the best particle position p_k^i for each particle, and the best swarm position p_k^g.
4. Update the position of each particle using its previous position and updated velocity vector (while enforcing bounds).
5. Repeat steps 2–4 until the stopping criteria is met.

A number of different stopping criteria are possible:

- The best fitness value has not changed for several iterations (a relatively large number of iterations because the best value changes much less frequently than other metrics).
- the mean of the best fitness values (mean of p_k^i) of the swarm does not change (within some tolerance) for a number of iterations.
- the velocity of the particles in the swarm falls below some tolerance for a number of iterations.
- the distance between every particle and the best particle is below some tolerance.
- the difference between the best and the worst fitness is below some tolerance.

Velocity clamping is often used in PSO. Velocity clamping means that a maximum velocity is imposed so that speeds cannot diverge. The maximum speed is usually scaled based on the bounds:

$$V_{maxi} = \delta(x_{maxi} - x_{mini}) \quad \text{with} \quad \delta \in (0, 1) \tag{6.27}$$

PSO Characteristics

Compared to other global optimization approaches PSO:

- Is a simple algorithm that is easy to implement.
- Is still a population based algorithm, however it works well with few particles (10 to 40 are usual) and does not have "generations".
- Is unlike evolutionary approaches in that design variables are directly updated, and there are no chromosomes, survival of the fittest, selection or crossover operations.
- Has global and local search behavior that can be directly "adjusted" as desired using the cognitive c_1 and social c_2 parameters.
- Allows for convergence "balance" through the inertial weight factor w

Algorithm Analysis

If we replace the velocity update equation into the position update the following expression is obtained:

$$x_{k+1}^i = x_k^i + \left(w v_k^i + c_1 r_1 \frac{\left(p_k^i - x_k^i\right)}{\Delta t} + c_2 r_2 \frac{\left(p_k^g - x_k^i\right)}{\Delta t} \right) \Delta t \tag{6.28}$$

Factorizing the cognitive and social terms:

$$x_{k+1}^i = \underbrace{x_k^i + w v_k^i \Delta t}_{\hat{x}_k^i} + \underbrace{(c_1 r_1 + c_2 r_2)}_{\alpha_k} \underbrace{\left(\frac{c_1 r_1 p_k^i + c_2 r_2 p_k^g}{c_1 r_1 + c_2 r_2} - x_k^i \right)}_{\hat{p}_k} \tag{6.29}$$

So the behavior of each particle can be viewed as a line-search dependent on a stochastic step size and search direction.

Re-arranging the position and velocity term in the above equation we have:

$$\begin{array}{l} x_{k+1}^i = x_k^i \left(1 - c_1 r_1 - c_2 r_2\right) + w V_k^i \Delta t + c_1 r_1 p_k^i + c_2 r_2 p_k^g \\ v_{k+1}^i = -x_k^i \frac{(c_1 r_1 + c_2 r_2)}{\Delta t} + w V_k^i + c_1 r_1 \frac{p_k^i}{\Delta t} + c_2 r_2 \frac{p_k^g}{\Delta t} \end{array} \tag{6.30}$$

which can be combined and written in a matrix form as:

$$\begin{bmatrix} x_{k+1}^i \\ V_{k+1}^i \end{bmatrix} = \begin{bmatrix} 1 - c_1 r_1 - c_2 r_2 & w \Delta t \\ -\frac{(c_1 r_1 + c_2 r_2)}{\Delta t} & w \end{bmatrix} \begin{bmatrix} x_k^i \\ V_k^i \end{bmatrix} + \begin{bmatrix} c_1 r_1 & c_2 r_2 \\ \frac{c_1 r_1}{\Delta t} & \frac{c_2 r_2}{\Delta t} \end{bmatrix} \begin{bmatrix} p_k^i \\ p_k^g \end{bmatrix} \tag{6.31}$$

where the above representation can be seen as a representation of a discrete-dynamic system from which we can find stability criteria [10].

Assuming constant external inputs, the system reduces to:

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -\left(c_1 r_1 + c_2 r_2\right) & w \Delta t \\ -\frac{(c_1 r_1 + c_2 r_2)}{\Delta t} & w - 1 \end{bmatrix} \begin{bmatrix} x_k^i \\ V_k^i \end{bmatrix} + \begin{bmatrix} c_1 r_1 & c_2 r_2 \\ \frac{c_1 r_1}{\Delta t} & \frac{c_2 r_2}{\Delta t} \end{bmatrix} \begin{bmatrix} p_k^i \\ p_k^g \end{bmatrix} \tag{6.32}$$

where the above is true only when $V_k^i = 0$ and $x_k^i = p_k^i = p_k^g$ (equilibrium point).

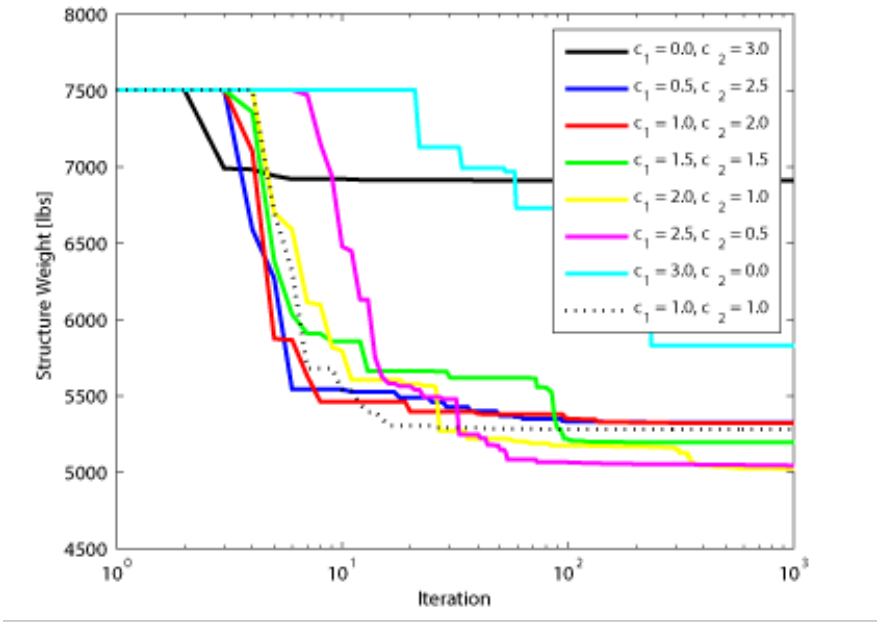
The eigenvalues of the dynamic system are:

$$\lambda^2 - \left(w - c_1 r_1 - c_2 r_2 + 1\right) \lambda + w = 0 \tag{6.33}$$

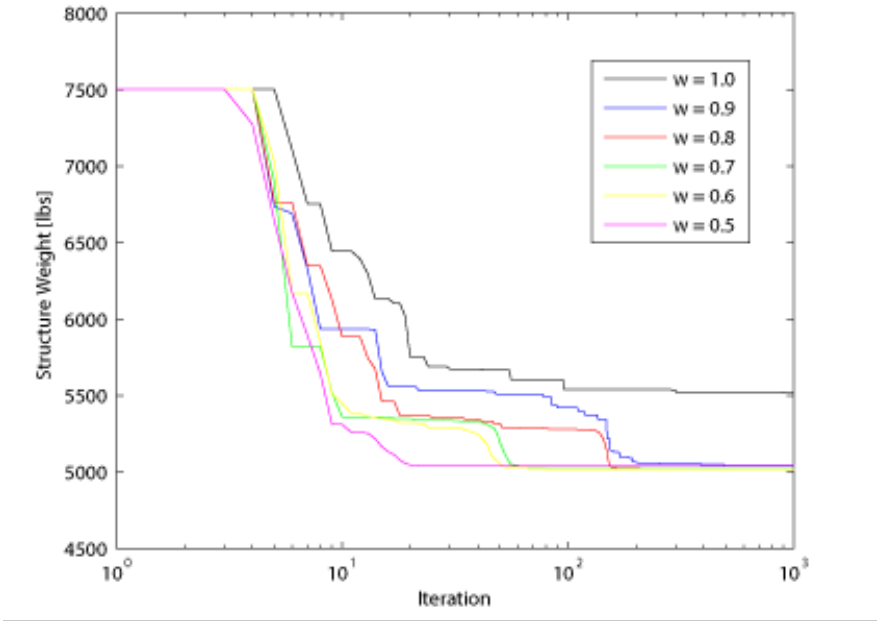
Hence, the stability in the PSO dynamic system is guaranteed if $|\lambda_{i=1,\ldots,n}| < 1$, which leads to:

$$\begin{array}{l} 0 < (c_1 + c_2) < 4 \\ \frac{(c_1 + c_2)}{2} - 1 < w < 1 \end{array} \tag{6.34}$$

Effect of varying c_1 and c_2:

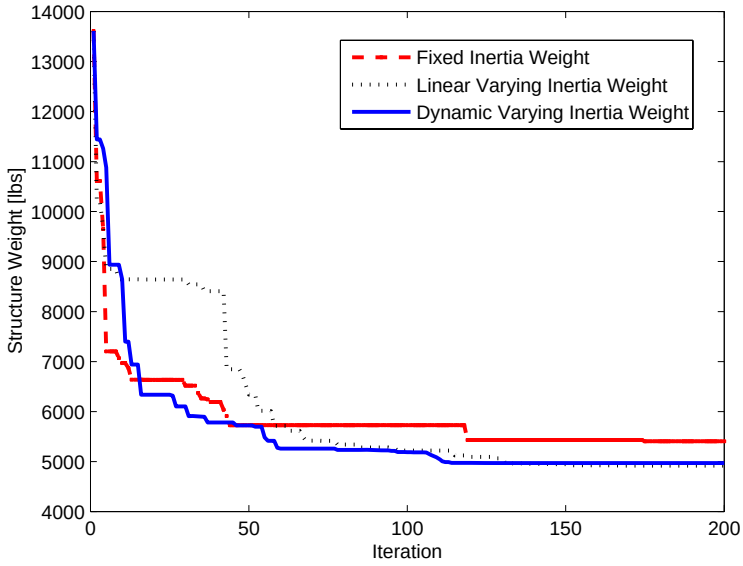


Effect of varying the inertia:



Algorithm Issues and Improvements
Updating the Inertia Weight:

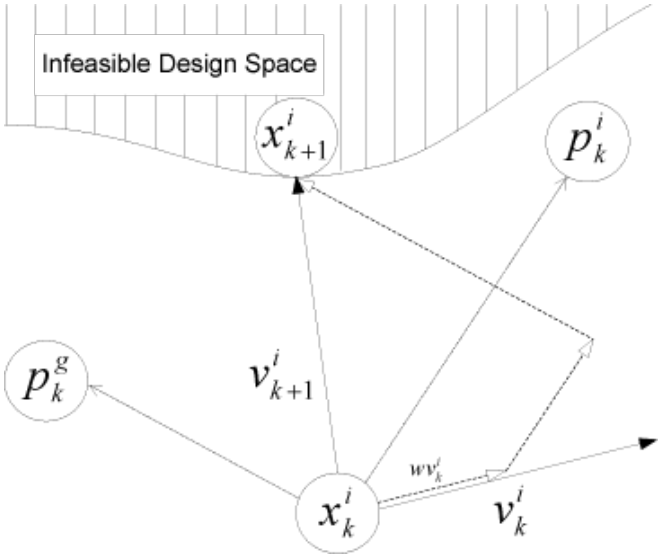
- As $k \to \infty$ particles "cluster" towards the "global" optimum.
- Fixed inertia makes the particles to overshoot the best regions (too much momentum).
- A better way of controlling the global search is to dynamically update the inertia weight.

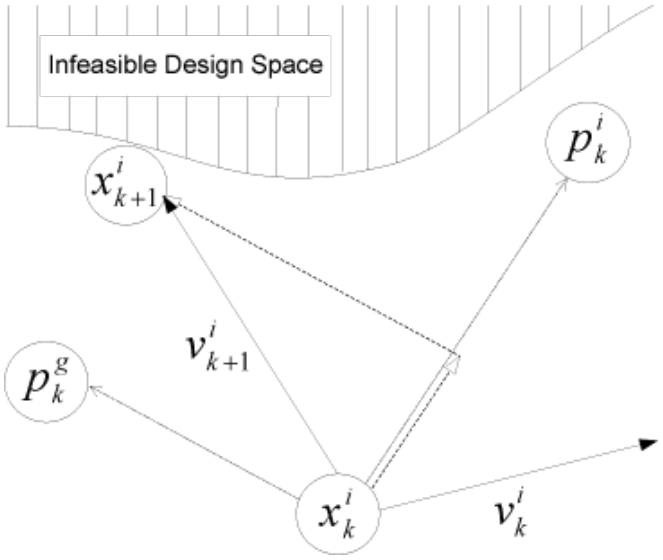


Violated Design Points Redirection:

We can restrict the velocity vector of a constraint violated particle to a usable feasible direction:

$$v_{k+1}^{i} = c_1 r_1 \frac{\left(p_k^i - x_k^i\right)}{\Delta t} + c_2 r_2 \frac{\left(p_k^g - x_k^i\right)}{\Delta t} \tag{6.35}$$





Constraint Handling:
The basic PSO algorithm is an unconstrained optimizer, to include constraints we can use:

- Penalty Methods
- Augmented Lagrangian function

Recall the Lagrangian function:

$$\mathcal{L}_i\left(x_k^i, \lambda^i\right) = f\left(x_k^i\right) + \sum_{j=1}^{m} \lambda_j^i g_j\left(x_k^i\right) \tag{6.36}$$

Augmented Lagrangian function:

$$\mathcal{L}_i\left(x_k^i, \lambda^i, r_p^i\right) = f\left(x_k^i\right) + \sum_{j=1}^{m} \lambda_j^i \theta_j\left(x_k^i\right) + \sum_{j=1}^{m} r_{p,j} \theta_j^2\left(x_k^i\right) \tag{6.37}$$

where:

$$\theta_j\left(x_k^i\right) = \max\left[g_j\left(x_k^i\right), \frac{-\lambda_j}{2r_{p,i}}\right] \tag{6.38}$$

- Multipliers and penalty factors that lead to the optimum are unknown and problem dependent.
- A sequence of unconstrained minimizations of the Augmented Lagrange function are required to obtain a solution.

Multiplier Update

$$\lambda_j^i\big|_{v+1} = \lambda_j^i\big|_v + 2\, r_{p,j}|_v\, \theta_j\left(x_k^i\right) \tag{6.39}$$

Penalty Factor Update (Penalizes infeasible movements):

$$r_{p,j}|_{v+1} = \begin{cases} 2\, r_{p,j}|_v & \text{if} \quad g_j\left(x_v^i\right) > g_j\left(x_{v-1}^i\right) \wedge g_j\left(x_v^i\right) > \varepsilon_g \\ \frac{1}{2} r_{p,j}|_v & \text{if} \quad g_j\left(x_v^i\right) \leq \varepsilon_g \\ r_{p,j}|_v & \text{otherwise} \end{cases} \tag{6.40}$$

Augmented Lagrangian PSO Algorithm [7]:

1. Initialize a set of particles positions x_o^i and velocities v_o^i randomly distributed throughout the design space bounded by specified limits. Also initialize the Lagrange multipliers and penalty factors, e.g. $\lambda_j^i\Big|_0 = 0,\ r_{p,j}|_0 = r_0$. Evaluate the objective function values using the initial design space positions.
2. Solve the unconstrained optimization problem described in the Augmented Lagrange Multiplier Equation using the basic PSO algorithm for $k_{\max}$ iterations.
3. Update the Lagrange multipliers and penalty factors.
4. Repeat steps 2–4 until a stopping criterion is met.

Example 6.36. Minimizing the Griewank Function

So how do the different gradient-free methods compare? A simple (but challenging!) numerical example is the Griewank Function for $n = 100$,

$$lf(x) = \sum_{i=1}^{n} \left(\frac{x_i^2}{4000} \right) - \prod_{\mathrm{i}=1}^{\mathrm{n}} \cos\left(\frac{\mathrm{x_i}}{\sqrt{\mathrm{i}}} \right) + 1 \tag{6.41}$$

$$-600 \leq \mathrm{x_i} \leq 600 \tag{6.42}$$

$$\tag{6.43}$$

Optimizer	Evaluations	Global optimum?	Objective	CPU time (s)
PSO (pop 40)	12,001	Yes	6.33e-07	15.9
GA (pop 250)	51,000	No	86.84	86.8438
DIRECT	649,522	Yes	1.47271e-011	321.57

Table 6.1: Comparison of gradient-free methods applied the minimization of the Griewank function. The results of PSO and GA are the mean of 50 trials.

Bibliography

[1] Nicolas E. Antoine and Ilan M. Kroo. Aircraft optimization for minimal environmental impact. *Journal of Aircraft*, 41(4):790–797, 2004.

[2] Ashok D. Belegundu and Tirupathi R. Chandrupatla. *Optimization Concepts and Applications in Engineering*, chapter 7. Prentice Hall, 1999.

[3] Andrew R. Conn, Katya Scheinberg, and Luis N. Vicente. *Introduction to Derivative-Free Optimization*. SIAM, 2008.

[4] J. E. Dennis and Virginia J. Torczon. Derivative-free pattern search methods for multidisciplinary design problems. *AIAA Paper* 94-4349, 1994.

[5] R. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *6th International Symposium on Micro Machine and Human Science*, number 0-7803-2676-8, May 1995.

[6] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*, chapter 1. Addison-Wesley Longman Publishing, Boston, MA, 1989.

[7] Peter W. Jansen and Ruben E. Perez. Constrained structural design optimization via a parallel augmented Lagrangian particle swarm optimization approach. *Computers & Structures*, 89: 1352–1366, 2011. doi:10.1016/j.compstruc.2011.03.011.

[8] J. A. Nelder and R. Mead. Simplex method for function minimization. *Computer Journal*, 7: 308–313, 1965.

[9] Chinyere Onwubiko. *Introduction to Engineering Design Optimization*, chapter 4. Prentice Hall, 2000.

[10] R. Perez and K. Behdinan. Particle swarm approach for structural design optimization. *International Journal of Computer and Structures*, 85(19-20):1579–1588, October 2007.

[11] C.D. Perttunen, D.R. Jones, and B.E. Stuckman. Lipschitzian optimization without the lipschitz constant. *Journal of Optimization Theory and Application*, 79(1):157–181, October 1993.

[12] David J. Powell, Siu Shing Tong, and Michael M. Skolnick. Engeneous domain independent, machine learning for design optimization. In *Proceedings of the third international conference on Genetic algorithms*, pages 151–159, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[13] Peter Sturdza. An aerodynamic design method for supersonic natural laminar flow aircraft. PhD thesis 96–98, Stanford University, Stanford, California, 2003.

Chapter 7

Multidisciplinary Design Optimization

7.1 Introduction

In the last few decades, numerical models that predict the performance of engineering systems have been developed, and many of these models are now mature areas of research. For example, finite element structural analysis is now routine in industry, as is computational fluid dynamics for certain cases. Once engineers are able to model a system and predict the effect that changes in the design have on the performance of that system, the next logical question is what changes in the design produced optimal performance. The application of the numerical optimization techniques described in the preceding chapters address this question.

While single-discipline optimization is in some cases quite mature, the design and optimization of systems that involve more than one discipline is still in its infancy. When systems are composed of multiple systems, additional issues arise in both the analysis and design optimization, and hence the present chapter.

Multidisciplinary design optimization (MDO) is a field of engineering that focuses on use of numerical optimization to perform the design of systems that involve a number of disciplines or subsystems. The main motivation for using MDO is that the performance of a multidisciplinary system is driven not only by the performance of the individual disciplines, but also by their interactions. Considering these interactions in an optimization problem generally requires a sound mathematical formulation. By solving the MDO problem early in the design process and taking advantage of advanced computational analysis tools, designers can simultaneously improve the design and reduce the time and cost of the design cycle.

MDO is a field of growing recognition in both academic and industrial circles. Although there have been hundreds of research papers written on the subject, MDO has still not lived up to its full potential. Over the last few decades, MDO has been increasingly used in the aircraft design industry. The increase in the complexity of aircraft has made this a more challenging proposition, but together with the demand for better performing machines, it has made it a necessity.

The origins of MDO can be traced back to Schmit [139, 140, 141] and Haftka [57, 58, 60], who extended their experience in structural optimization to include other disciplines. One of the first applications of MDO was aircraft wing design, where aerodynamics, structures, and controls are three strongly coupled disciplines [53, 54, 104, 105]. Since then, the application of MDO has been extended to complete aircraft [93, 107] and a wide range of other engineering systems, such as bridges [9], buildings [34, 49], railway cars [44, 66], microscopes [132], automobiles [90, 116], ships [73, 131], rotorcraft [47, 51], and spacecraft [22, 29].

The reason why MDO was pioneered by aerospace engineering researchers is not coincidental.

Aircraft are prime examples of multidisciplinary systems; for aircraft to fulfil their function, the disciplines of aerodynamics, structures, propulsion and control all come into play. Before the dawn of aviation, Sir George Cayley recognized the multidisciplinary nature of flying machines, and proposed the separation of aircraft components according to function, which resulted in the same components we still see today: wing to create lift, fuselage to carry payload, and tail to provide control. Cayley envisioned that these different component would be designed separately while ignoring their mutual interactions, and integrated at a later stage of development. This multidisciplinary segregated approach served the Wright brothers well, and is still used today.

However, over one hundred years later, aircraft have become much more complex engineering systems and the expectations on their performance have increased continuously to levels that early designers could only dream of. This increased expectation of performance, has lead engineers to question the separation of component design, disciplines, or both, and consider a more integrative approach. The basic principle of multidisciplinary design optimization (MDO) is that a given component or discipline affects all others, and therefore, one must consider this coupling in the design process to take advantage of beneficial interactions.

Richard Whitcomb's supercritical wing [**?**] provides an example of how important it is to consider the various disciplines involved when making design decisions. When Whitcomb devised a shape that reduced the shock strength and thus the wave drag, he assumed that future transport airplanes using this technology would fly at higher subsonic Mach numbers. However, after engineers considered the multidisciplinary trade-offs, they realized that the supercritical wing could have less sweep, be made thicker, or both, for same cruise Mach number without a drag penalty. Both lower sweep and larger wing thickness reduce the structural weight, which ultimately reduces the fuel burn.

In the opinion of some MDO researchers, industry will not adopt MDO more widely because they do not realize their utility. Some engineers in industry think that researchers are not preaching anything new, since industry has already been doing multidisciplinary design. There is some truth to each of these perspectives. Real-world aerospace design problem may involve thousands of variables and hundreds of analyses and engineers, and it is often difficult to apply the numerical optimization techniques and solve the mathematically correct optimization problems.The kinds of problems in industry are often of much larger scale, involve much uncertainty, and include human decisions in the loop, making them difficult to solve with traditional numerical optimization techniques. On the other hand, a better understanding of MDO by engineers in industry is now contributing a more widespread use in practical design.

An overview of the structure of an aircraft company is shown in Fig. 7.1. We can see that the personnel is organized in a neat hierarchy with multiple levels, where the communication between groups of the same level is only supposed to take place through between the managers of the groups and other higher levels. However, the reality is that the design process requires a lot more communication, as shown on the right, and the process is incompatible with the way the groups are organized.

MDO is concerned with the development of strategies that utilize current numerical analyses and optimization techniques to enable the automation of the design process of a multidisciplinary system. One of the big challenges is to make sure that such a strategy is scalable and that it will work in realistic problems. An MDO *architecture* is a particular strategy for organizing the analysis software, optimization software, and optimization subproblem statements to achieve an optimal design.

In the MDO literature, there are several terms used to describe what we mean by "architecture": "method"[1, 93, 136, 154, 167], "methodology"[82, 117, 122], "problem formulation"[4, 5, 37], "strategy"[61, 170], "procedure"[85, 148] and "algorithm"[43, 143, 146, 153] have all been used.

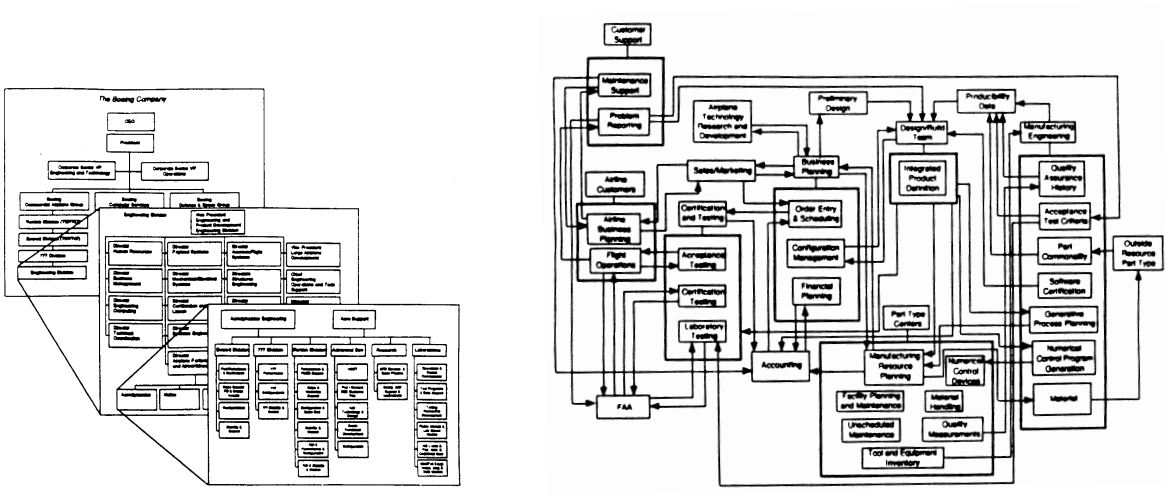


Figure 7.1: Flow charts illustrating the hierarchical organization of the personnel in an aircraft company (left) and the design process in the same company (right). Although the personnel is very structured, the design process is more chaotic and needs to cut across the hierarchy.

Some authors use a variety of terms and occasionally use them interchangeably in the same paper. Our preference for the term "architecture" [24, 32, 92, 147] comes from the fact that the relationship between problem formulation and solution algorithm is not one-to-one. For example, replacing a particular disciplinary simulation with a surrogate model or reordering the disciplinary simulations do not affect the problem formulation but strongly affect the solution algorithm.

7.2 Nomenclature and Mathematical Notation

In addition to the usual nomenclature we have used thus far, such as f for the objective function and x for the design variables, and c for the vector of constraints, we will use another set of variables and appropriate subscripts and superscripts to distinguish between subsets.

The notation is listed in Table 7.1. This is not a comprehensive list; additional notation specific to particular architectures is introduced when the respective architectures are described. We also take this opportunity to clarify many of the terms we use that are specific to the field of MDO.

In MDO, we make the distinction between *local* design variables, which directly affect only one discipline, and *shared* design variables, which directly affect more than one discipline. We denote the vector of design variables local to discipline i by x_i and shared variables by x_0. The full vector of design variables is given by $x = \left[x_0^T, x_1^T, \ldots, x_N^T\right]^T$. The subscripts for local and shared data are also used in describing objectives and constraints. The set of constraints are similarly split into those that are shared between disciplines and those that are discipline-specific, i.e. those which require only local variable information.

A *discipline analysis* is a simulation that models the behavior of one aspect of a multidisciplinary system. Running a discipline analysis consists in solving a system of equations — such as the Navier–Stokes equations in fluid mechanics, the static equilibrium equations in structural mechanics, or the equations of motion in a control simulation — which compute a set of discipline responses, known as *state variables*. State variables may or may not be controlled by the optimization, depending on the formulation employed. We denote the vector of state variables computed

Table 7.1: Mathematical notation for MDO problem formulations

Symbol	Definition
x	Vector of design variables
y^t	Vector of coupling variable targets (inputs to a discipline analysis)
y	Vector of coupling variable responses (outputs from a discipline analysis)
$\bar{y}$	Vector of state variables (variables used inside only one discipline analysis)
f	Objective function
c	Vector of design constraints
c^c	Vector of consistency constraints
$\mathcal{R}$	Governing equations of a discipline analysis in residual form
N	Number of disciplines
$n_{()}$	Length of given variable vector
$m_{()}$	Length of given constraint vector
$()_0$	Functions or variables that are shared by more than one discipline
$()_i$	Functions or variables that apply only to discipline i
$()^*$	Functions or variables at their optimal value
$\tilde{()}$	Approximation of a given function or vector of functions
$\hat{()}$	Duplicates of certain variable sets distributed to other disciplines

within discipline i by $\bar{y}_i$. We denote the associated set of disciplinary equations in residual form by $\mathcal{R}_i$, so that the expression $\mathcal{R}_i = 0$ represents the solution of these equations with respect to $\bar{y}_i$.

In a multidisciplinary system, most disciplines are required to exchange *coupling variables* to model the interactions of the whole system. Often, the number of variables exchanged is much smaller than the total number of state variables computed in a particular discipline. For example, in aircraft design, state information about the entire flow field resulting from the aerodynamics analysis is not required by the structural analyses. Only the aerodynamic loads on the aircraft surface are passed. The coupling variables supplied by a given discipline i are denoted by y_i. Another common term for y_i is *response variables*, since they describe the response of the analysis to a design decision. In general, a transformation is required to compute y_i from $\bar{y}_i$ for each discipline [37]. Similarly, a transformation may be needed to convert input coupling variables into a usable format within each discipline [37]. In this work, the mappings between y_i and $\bar{y}_i$ are lumped into the analysis equations $\mathcal{R}_i$. This simplifies our notation with no loss of generality.

In many formulations, *independent copies* of the coupling variables must be made to allow discipline analyses to run independently and in parallel. These copies are also known as *target variables*, which we denote by a superscript t. For example, the copy of the response variables produced by discipline i is denoted y_i^t. These variables are independent of the corresponding original variables and are used as part of the input to disciplines that are coupled to discipline i through y_i. In order to preserve consistency between the coupling variable inputs and outputs at the optimal solution, we define a set of *consistency constraints*, $c_i^c = y_i^t - y_i$, which we add to the optimization problem formulation.

Example 7.37. Aerostructural Design Optimization — Problem Definition

Throughout this chapter, we will use one example to illustrate the notation and MDO architectures. Suppose we want to design the wing of a business jet using low-fidelity analysis tools. We could, for example, model the aerodynamics using a simple panel method and represent the structure as a single beam composed of finite elements as shown in Fig. 7.2.

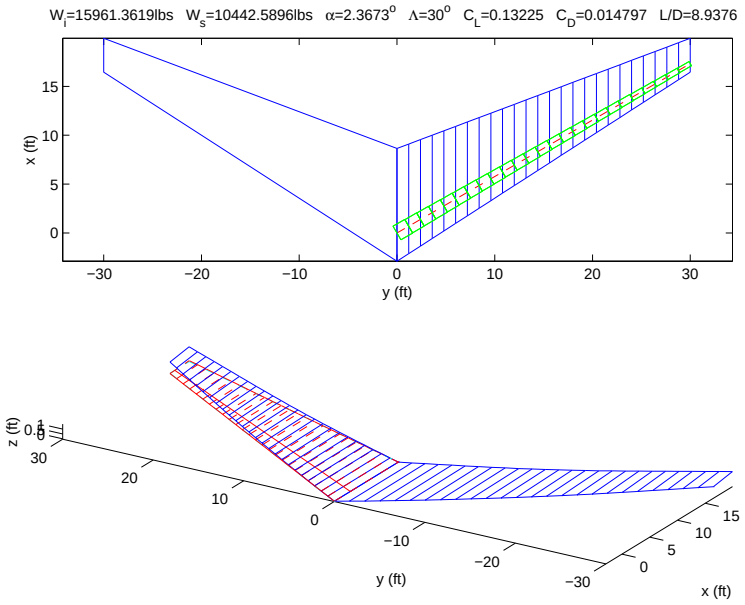


Figure 7.2: Low-fidelity aerostructural analysis model of a business jet wing.

The panel method takes as an input the angle-of-attack (α) a twist distribution (γ_i) and computes the lift (L) and the induced drag (D). The structural analysis takes the thicknesses of the beam (t_i) and computes the structural weight, which is added to a fixed weight to obtain the total weight (W). The maximum stresses in each finite-element (σ_i) are also calculated.

In this example, we want to maximize the range of the aircraft, as given by the Breguet range equation,

$$f = \text{Range} = \frac{V}{c}\frac{L}{D}\ln\left(\frac{W_i}{W_f}\right). \tag{7.1}$$

The multidisciplinary analysis consists in the simultaneous solution of the following equations:

$$\mathcal{R}_1 = 0 \Rightarrow \quad A\Gamma - v(u,\alpha) = 0 \tag{7.2}$$
$$\mathcal{R}_2 = 0 \Rightarrow \quad Ku - F(\Gamma) = 0 \tag{7.3}$$
$$\mathcal{R}_3 = 0 \Rightarrow \quad L(\Gamma) - W = 0 \tag{7.4}$$

The first equation represents the aerodynamic analysis, where A is the matrix of aerodynamic influence coefficients of the panel code, Γ is the vector of circulations, and v represents the panel flow tangency boundary conditions.

The second equation is the linear structural analysis, where K is the stiffness matrix, u are the displacement, and F are the external applied forces.

The third equation ensures that the lift is equal to weight at the flight condition.

Thus, the state for all disciplines for this case is,

$$y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} \Gamma \\ u \\ \alpha \end{bmatrix}. \tag{7.5}$$

The angle of attack is considered a state variable here, and helps satisfy $L = W$.

As an example, the design variables could be the wing sweep (Λ), structural thicknesses (t) and twist distribution (γ).

$$x_0 = \Lambda \tag{7.6}$$

$$x = \begin{bmatrix} t \\ \gamma \end{bmatrix}, \tag{7.7}$$

The sweep is a shared variable because changing the sweep has a direct effect on both the aerodynamic influence matrix and the stiffness matrix. The other two sets of design variables are local to the structures and aerodynamics, respectively.

In later sections, we will see the options we have to optimize the wing in this example.

7.3 Multidisciplinary Analysis

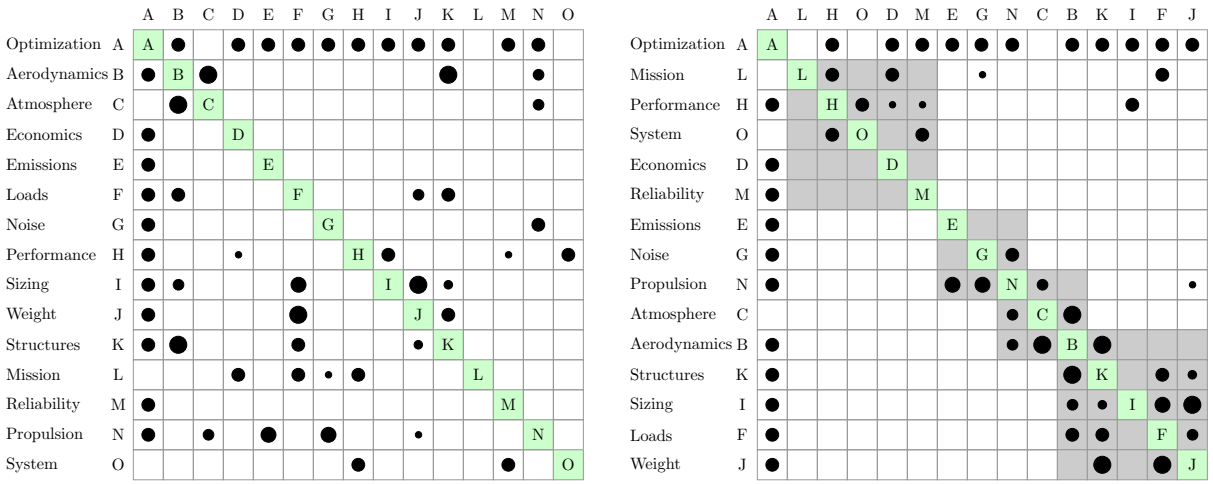
If we want to know how the system behaves for a given set of design variables without optimization, we need to repeat each disciplinary analysis until each $y_i^t = y_i^r$. This is called a *multidisciplinary analysis* (MDA). There are many possible techniques for converging the MDA but for now we will assume that the MDA is converged by a sequential (Gauss–Seidel) approach.

Algorithm 15 Block Gauss–Seidel multidisciplinary analysis algorithm

Input: Design variables x
Output: Coupling variables, y
0: Initiate MDA iteration loop
repeat
1: Evaluate Analysis 1 and update $y_1(y_2, y_3)$
2: Evaluate Analysis 2 and update $y_2(y_1, y_3)$
3: Evaluate Analysis 3 and update $y_3(y_1, y_2)$
until $4 \rightarrow 1$: MDA has converged

To visualize the interconnections between the various components of a system (disciplines in the case of MDO), systems engineering researchers introduced the design structure matrix (DSM) [27, 155]. An example of a DSM for an aircraft analysis problem is shown in Fig. 7.3. The circles in the upper right of the matrix show feed-forward connections, i.e., data that goes from the diagonal to the left of the circle to the diagonal below the circle. The circles in the lower left of the matrix show feedback connections that flow from the diagonal to the right of the circle to the diagonal above the circle.

As previously mentioned, one option to solve such a multidisciplinary system is through a fixed-point iteration, such as the Gauss–Seidel algorithm that we previously mentioned. However, this type of iteration tends to converge slowly and sometimes fails. One way to improve the Gauss–Seidel iteration is to reorder the disciplines in the matrix. An improved reordering will reduce the feedback connections as much as possible, and cluster highly connected disciplines, as shown in Fig. 7.3 on the right. The clusters are shown shaded in gray. Each of these clusters can be solved using Gauss–Seidel iteration one at the time, and it is not necessary to repeat the iteration of each cluster. In this case, we just need to perform the Gauss–Seidel iteration of the four clusters in sequence for the system to converge, resulting in an overall reduction in computational cost. In later sections, we will discuss more sophisticated methods for multidisciplinary analysis.



(a) Original ordering

(b) Improved ordering

Figure 7.3: DSM of an aircraft analysis and optimization; the size of the circles is proportional to the strength of the coupling.

7.4 Architecture Diagrams — The Extended Design Structure Matrix

While rewriting problem formulations for each architecture using a common notation is a straightforward task, describing the sequence of operations in the implementation in a convenient way presents a significant challenge. Some authors just present the problem formulation and let the readers work out the implementation by themselves. This is acceptable for monolithic architectures. For some of the distributed architectures, however, the implementation is not obvious. Other authors use an algorithm or flowchart as an aid, but these are often inadequate for describing the data flow between the different software components. Furthermore, more complex strategies with multiple loops and parallel processes are difficult to describe compactly using this technique. The lack of a standard convenient graphical representation to describe the solution strategy in MDO architectures is another impediment to understanding and comparing their relative merits.

To enhance our descriptions, each of the architectures in this chapter is presented using a new diagram that we call the *extended design structure matrix*, or XDSM [94]. As the name suggests, the XDSM was based on DSM, which was introduced above. The traditional DSM shows components and connections between components, but the meaning of the connections is left ambiguous. For the purposes of representing MDO architectures, we need two types of connections: data flow and process flow. These two types of connections are often fused together in diagrams, but as we will see, there is a need to distinguish between data dependency and process flow to correctly represent MDO architectures. This need motivated the development of XDSM, which simultaneously communicates data dependency and process flow between computational components of MDO architectures on a single diagram. Herein, we present only a brief overview of the XDSM. Further details on the XDSM and its applications are presented by Lambe and Martins [94].

We explain the basics of the XDSM using two simple examples. The first example is shown in Fig. 7.4, and represents a Gauss–Seidel multidisciplinary analysis (MDA) procedure for three disciplines. The pseudocode for this procedure is listed in Algorithm 15. As with the traditional DSM, the components are laid out along the diagonal. The components in this case consist of the discipline analyses and a special component, known as a *driver*, which controls the iteration and

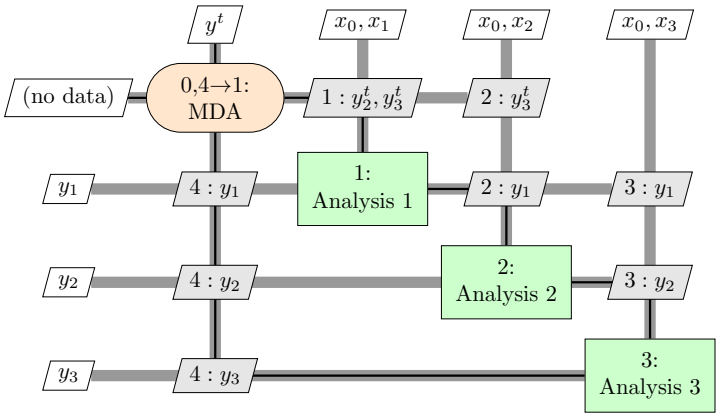


Figure 7.4: A block Gauss–Seidel multidisciplinary analysis (MDA) process to solve a three-discipline coupled system.

is represented by a rounded rectangle. The function of the components it to process data. The data flow is shown as thick gray lines. Components take data inputs from the vertical direction and output data in the horizontal direction. Thus, the connections in the upper diagonal flow from left to right and top to bottom, and the connections in the lower diagonal flow from right to left and bottom to top. The off-diagonal nodes in the shape of parallelograms are used to label the data. Using this convention, it is easy to identify the inputs of a given component by scanning the column above and below the component, while the outputs can be identified by scanning the row. External inputs and outputs are placed on the outer edges of the diagram — in the top row and leftmost column, respectively. In the case of Fig. 7.4, the external inputs are the design variables and an initial guess of the system coupling variables. Each discipline analysis computes its own set of coupling variables that is passed to other discipline analyses or back to the driver. At the end of the MDA process, each discipline returns the final set of coupling variables computed.

The thin black lines show the process flow. The directionality of these lines follows the same convention as the data flow lines. In addition, a numbering system is used to show the order in which the components are executed. These numbers are presented inside each component in the diagram followed by a colon and the component name. When tracing the algorithm, every time a number is reached, the corresponding component performs a relevant computation. Multiple numbers indicate that the component is called several times within the algorithm. The algorithm starts at component zero and proceeds in numerical order, following the process lines. Loops are denoted using the notation $j \rightarrow k$ for $k < j$ so that the algorithm must return to step k until some condition required by the driver is satisfied. The data nodes are also labeled with numbers to denote the time at which the input data is retrieved.

The second XDSM example, illustrated in Fig. 7.5, is the solution process for an analytic optimization problem using gradient-based optimization. The problem has a single objective and a vector of constraints. Fig. 7.5 shows separate components to compute the objective, constraints, and their gradients, as well as a driver to control the iteration. We assume for this problem that the gradients can be computed without knowledge of the objective and constraint function values. Notice that in this example, multiple components are evaluated at step one of the algorithm. This numbering denotes parallel execution. In some cases, it may be advisable to lump components together to reflect underlying problem structures, such as lumping together the objective and constraint components. In the following sections, we have done just that in the architecture diagrams

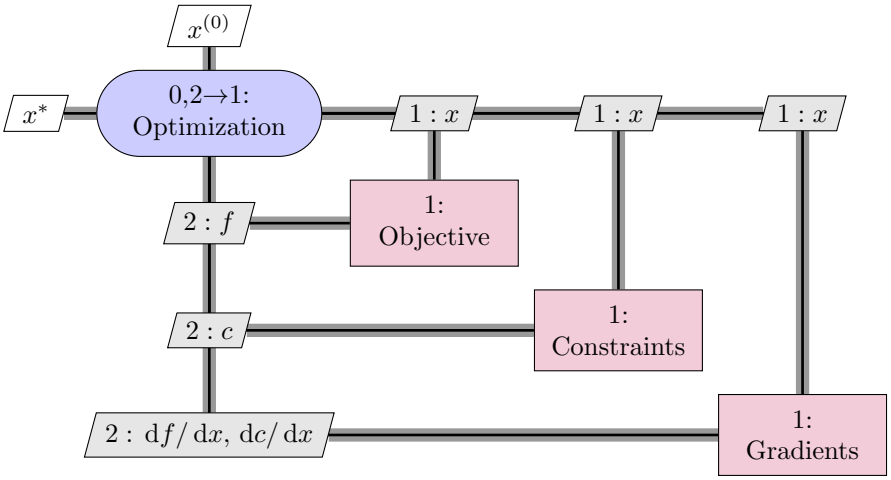


Figure 7.5: A gradient-based optimization procedure.

to simplify presentation. We also mention other simplifications as we proceed.

Example 7.38. Aerostructural Design Optimization — Sequential Optimization vs. MDO

Following up on the problem described in Ex. 37, we now consider how to optimize the range of the business jet by changing the wing twist distribution and sizing.

One intuitive approach would be to perform a *sequential optimization* approach. This consists in optimizing each discipline in sequence. For examle, we could start by optimizing the aerodynamics. Since the aerodynamic variables have no direct effect on the weight, maximizing the range is equivalent to minimizing cruise drag, and the problem can be stated as,

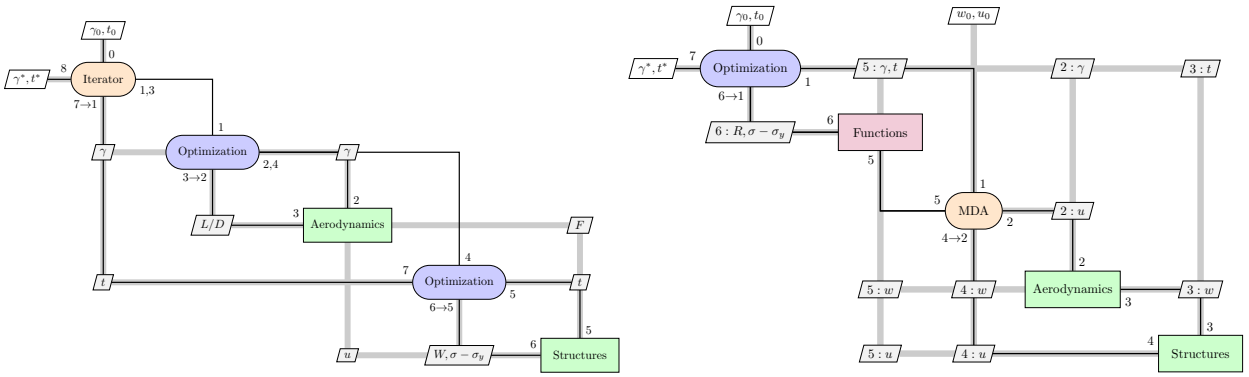
$$\begin{aligned} \text{minimize} \quad & D\left(\alpha, \gamma_i\right) && (7.8)\\ \text{w.r.t.} \quad & \alpha, \gamma_i && (7.9)\\ \text{s.t.} \quad & L\left(\alpha, \gamma_i\right) = W && (7.10) \end{aligned}$$

Once the aerodynamic optimization has converged, the twist distribution and the forces are fixed, and we then optimize the structure by minimizing weight subject to stress constraints at the maneuver condition, i.e.,

$$\begin{aligned} \text{minimize} \quad & W\left(t_i\right) && (7.11)\\ \text{w.r.t.} \quad & t_i && (7.12)\\ \text{s.t.} \quad & \sigma_j\left(t_i\right) \leq \sigma_{\text{yield}} && (7.13) \end{aligned}$$

And repeat until this sequence has converged. The XDSM for this procedure is shown on the left of Fig. 7.6.

The MDO procedure differs from the sequential approach in that it considers all variables



(a) Sequential (b) MDO

Figure 7.6: Procedure for aerostructural design

simultaneously, so that the optimization problem is,

$$\begin{aligned} \text{minimize} \quad & \text{Range}\,(\alpha, \gamma_i, t_i) && (7.14)\\ \text{w.r.t.} \quad & \alpha, \gamma_i, t_i && (7.15)\\ \text{s.t.} \quad & \sigma_{\text{yield}} - \sigma_j\,(t_i) \geq 0 && (7.16)\\ & L\,(\alpha, \gamma_i) - W = 0 && (7.17) \end{aligned}$$

The results of these two approaches are shown in Fig. 7.7. The sequential approach converges to an elliptical lift distribution that minimizes the induced drag, while the MDO approach converges to the compromise between aerodynamics and structures that yields the maximum range. The spanwise lift distribution in the MDO case moves some of the inboard, which reduces the moments in the wing and results in a lighter structure. The lighter weight more than compensates for the increased drag.

To further illustrate the difference between the sequential and MDO approaches, we solve this problem for two variables — one thickness and one twist — so that we can plot the convergence history for the two design variables overlaid on a contour plot of the range, as shown in Fig. 7.8

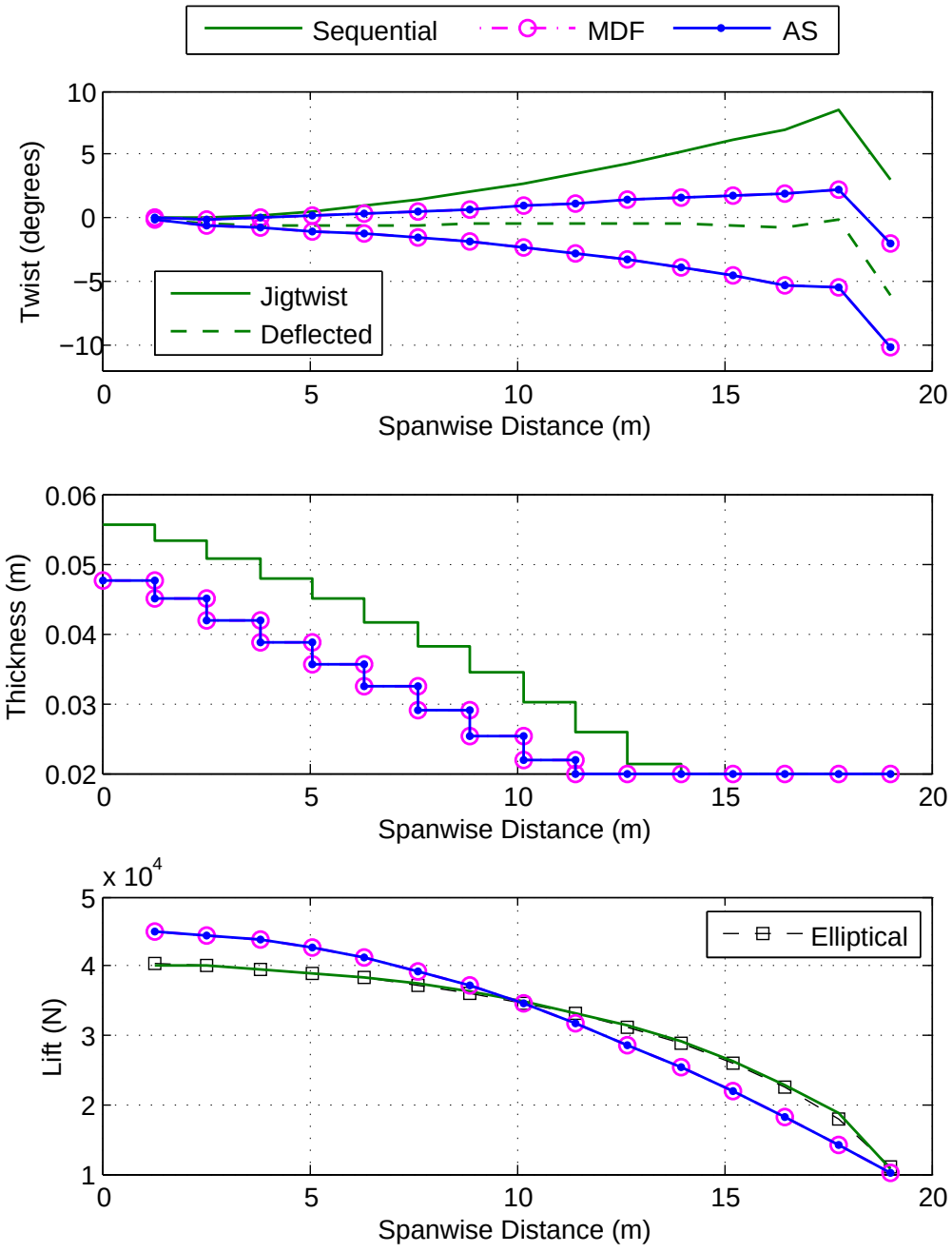


Figure 7.7: Spanwise distributions of twist, thickness and lift for sequential and MDO approaches

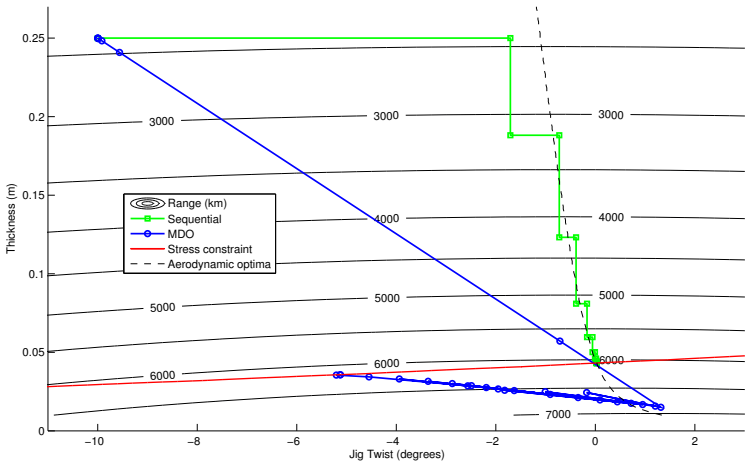


Figure 7.8: Contour plot of the range with the the paths followed by the sequential and MDO approaches. The sequential optimization converges to a non-optimal point, while the MDO approach converges to the true optimum.

7.5 Monolithic Architectures

If we ignore the disciplinary boundaries, an MDO problem is nothing more than a standard constrained nonlinear programming problem: it involves solving for the values of the design variables that maximize or minimize a particular design objective function, subject to design constraints. The choice of design objectives, design constraints, and even what variables to change in a given system is strictly up to the designer. The behavior of each component, or discipline, within the system is modeled using a discipline analysis. Each discipline analysis is usually available in the form of a computer program and can range in complexity from empirical curve-fit data to a highly detailed physics-based simulation.

One of the major challenges of MDO is how to address the coupling of the system under consideration. Like the disciplines they model, the discipline analyses themselves are mutually interdependent. A discipline analysis requires outputs of other analyses as input to resolve themselves correctly. Furthermore, the objective and constraint functions, in general, depend on both the design variables and analysis outputs from multiple disciplines. While this interdependence is sometimes ignored in practice through the use of single discipline optimizations occurring in parallel or in sequence, taking the interdependence into account generally leads to a more accurate representation of the behavior of the whole system. MDO architectures provide a consistent, formal setting for managing this interdependence in the design process [92].

The architectures presented in this section are referred to as *monolithic* architectures. Each architecture solves the MDO problem by casting it as single optimization problem. The differences between architectures lie in the strategies used to achieve multidisciplinary feasibility. Architectures that decompose the optimization problem into smaller problems, i.e., *distributed* architectures, are presented in Section 7.6.

7.5.1 The All-at-Once (AAO) Problem Statement

Before discussing specific architectures, we show the most fundamental optimization problem from which all other problem statements are derived. We can describe the MDO problem in its most

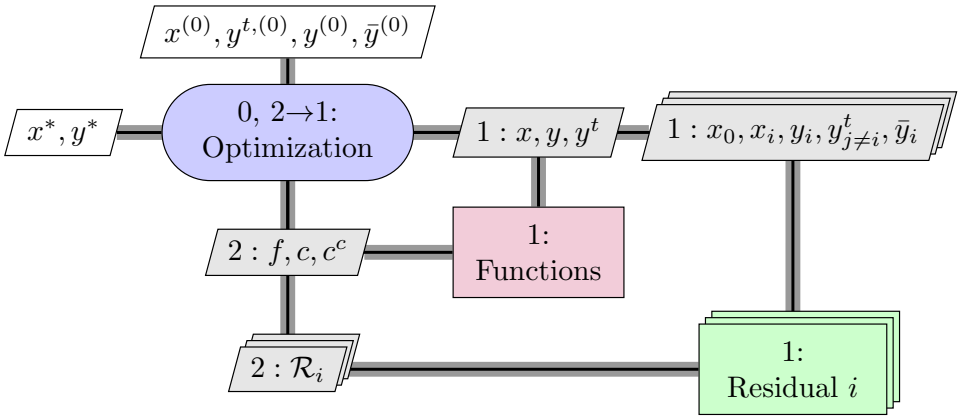


Figure 7.9: XDSM for solving the AAO problem.

general form as

$$
\begin{aligned}
\text{minimize} \quad & f_0(x,y) + \sum_{i=1}^{N} f_i(x_0, x_i, y_i) \\
\text{with respect to} \quad & x, y^t, y, \bar{y} \\
\text{subject to} \quad & c_0(x,y) \geq 0 \\
& c_i(x_0, x_i, y_i) \geq 0 \qquad \text{for} \quad i = 1, \ldots, N \\
& c^c_i = y^t_i - y_i = 0 \qquad \text{for} \quad i = 1, \ldots, N \\
& \mathcal{R}_i\left(x_0, x_i, y^t_{j\neq i}, \bar{y}_i, y_i\right) = 0 \qquad \text{for} \quad i = 1, \ldots, N.
\end{aligned}
\tag{7.18}
$$

For all design and state variable vectors, we use the notation $x = [x_0^T, x_1^T, \ldots, x_N^T]^T$ to concatenate the disciplinary variable groups. In future problem statements, we omit the local objective functions f_i except when necessary to highlight certain architectural features. Design constraints that are equalities can also be accomodated in Problem (7.18) without loss of generality by restating these constraints as a pair of inequalities with opposing signs. Problem (7.18) is known as the "all-at-once" (AAO) problem. Fig. 7.9 shows the XDSM for solving this problem. To keep the diagrams compact, we adopt the convention that any block referring to discipline i represents a repeated pattern for every discipline. Thus, in Fig. 7.9 a residual block exists for every discipline in the problem and each block can be executed in parallel. As an added visual cue in the XDSM, the "Residual i" component is displayed as a stack of similar components. There is a conflict with the established literature when it comes to the labeling of Problem (7.18). What most authors refer to as AAO, following the lead of Cramer et al. [37], others label as the simultaneous analysis and design (SAND) [10, 59] problem. Our AAO problem is most like what Cramer et al. refer to as simply "the most general formulation" [37]. Herein, we classify the formulation (7.18) as the AAO problem, because it includes all design, state, and input and output coupling variables in the problem, so the optimizer is responsible for all variables at once. The SAND architecture uses a different problem statement and is presented in Section 7.5.2.

The AAO problem is never solved in this form practice, because the consistency constraints, which are linear in this formulation, can be eliminated quite easily. Eliminating these constraints reduces the problem size without compromising the performance of the optimization algorithm.

As we will see, eliminating the consistency constraints from Problem (7.18) results in the problem solved by the SAND architecture. However, we have presented the AAO problem first because it functions as a common starting point for deriving both the SAND problem and the Individual Discipline Feasible (IDF) problem and, subsequently, all other equivalent MDO problems.

Depending on which equality constraint groups are eliminated from Problem (7.18), we can derive the other three monolithic architectures: Multidisciplinary Feasible (MDF), Individual Discipline Feasible (IDF), and Simultaneous Analysis and Design (SAND). All three have been known in the literature for a long time [10, 37, 59, 142]. In the next three subsections, we describe how each architecture is derived and the relative advantages and disadvantages of each. We emphasize that in all cases, in spite of the elements added or removed by each architecture, *we are always solving the same MDO problem.*

7.5.2 Simultaneous Analysis and Design (SAND)

The most obvious simplification of Problem (7.18) is to eliminate the consistency constraints, $c_i^c = y_i^t - y_i = 0$, by introducing a single group of the coupling variables to replace the separate target and response groups. This simplification yields the SAND architecture [59], which solves the following optimization problem:

$$\begin{aligned}
\text{minimize} \quad & f_0(x, y) \\
\text{with respect to} \quad & x, y, \bar{y} \\
\text{subject to} \quad & c_0(x, y) \geq 0 \\
& c_i(x_0, x_i, y_i) \geq 0 \qquad \text{for} \quad i = 1, \ldots, N \\
& \mathcal{R}_i(x_0, x_i, y, \bar{y}_i) = 0 \quad \text{for} \quad i = 1, \ldots, N.
\end{aligned} \tag{7.19}$$

The XDSM for SAND is shown in Fig. 7.10. Cramer et al. [37] refer to this architecture as "All-at-Once". However, we use the name SAND to reflect the consistent set of analysis and design variables chosen by the optimizer. The optimizer, therefore, can simultaneously analyze and design the system.

Several features of the SAND architecture are noteworthy. Because we do not need to solve any discipline analysis explicitly or exactly at each optimization iteration, the optimization problem can potentially be solved very quickly by letting the optimizer explore regions that are infeasible with respect to the analysis constraints, $\mathcal{R}_i$. The SAND methodology is not restricted to multidisciplinary systems and can be used in single discipline optimization as well. In that case, we only need to define a single group of design constraints, c. If the disciplinary residual equations are simply discretized partial differential equations, the SAND problem is just a PDE-constrained optimization problem like many others in the literature. (See Biegler et al. [16] for an overview of this field.)

Two major issues are still present in the SAND architecture. First, the problem formulation still requires all state variables and discipline analysis equations, meaning that large problem size and potential premature termination of the optimizer at an infeasible design can be issues in practice. Second, and more importantly, the fact that the discipline analyses equations are treated explicitly as constraints means that the residual values — and possibly their derivatives — need to be available to the optimizer. In other words, rather than computing coupling variables y_i and state variables $\bar{y}_i$, each discipline i accepts predetermined values of y_i and $\bar{y}_i$ and returns analysis equation residuals $\mathcal{R}_i$. In engineering design, many discipline analysis codes operate in a "black-box" fashion, directly computing the coupling variables while hiding the discipline analyses residuals and state variables in the process. Even if the source code for the discipline analysis can be modified to return

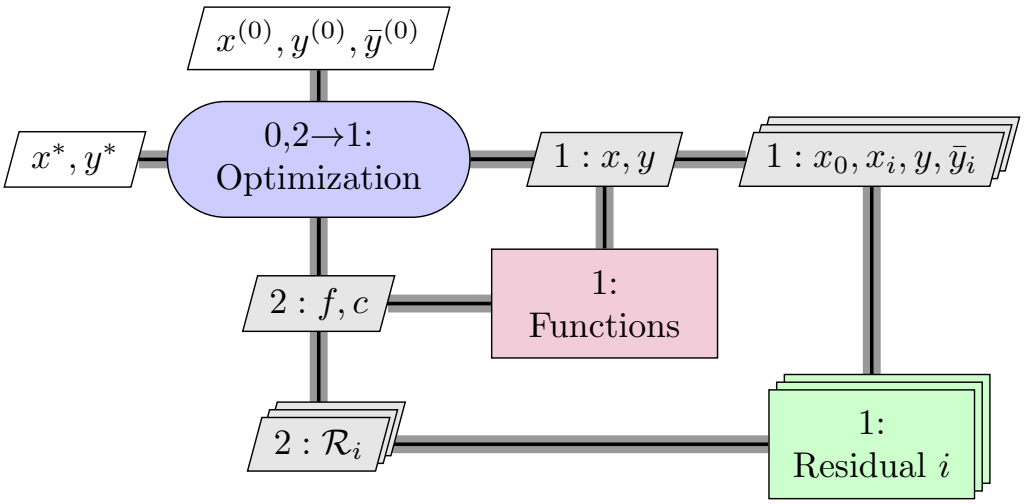


Figure 7.10: Diagram for the SAND architecture.

residuals, the cost and effort required often eliminates this option from consideration. Therefore, most practical MDO problems require an architecture that can take advantage of existing discipline analysis codes. The following two monolithic architectures address this concern.

Example 7.39. Aerostructural Optimization Using SAND

$$\begin{aligned}
\text{minimize} \quad & -R && (7.20)\\
\text{w.r.t.} \quad & \Lambda, \gamma, t, \Gamma, \alpha, u && (7.21)\\
\text{s.t.} \quad & \sigma_{\text{yield}} - \sigma_i(u) \geq 0 && (7.22)\\
& A\Gamma = v(u, \alpha) && (7.23)\\
& K(t)u = f(\Gamma) && (7.24)\\
& L(\Gamma) - W(t) = 0 && (7.25)
\end{aligned}$$

7.5.3 Individual Discipline Feasible (IDF)

By eliminating the disciplinary analysis constraints $\mathcal{R}_i(x_0, x_i, y_i, y^t_{j\neq i}, \bar{y}_i) = 0$ from Problem (7.18), we obtain the IDF architecture [37]. As commonly noted in the literature, this type of elimination is achieved by applying the Implicit Function Theorem to the $\mathcal{R}_i$ constraints so that $\bar{y}_i$ and y_i become functions of design variables and coupling targets. The IDF architecture is also known as distributed analysis optimization [4] and optimizer-based decomposition [92]. The optimization

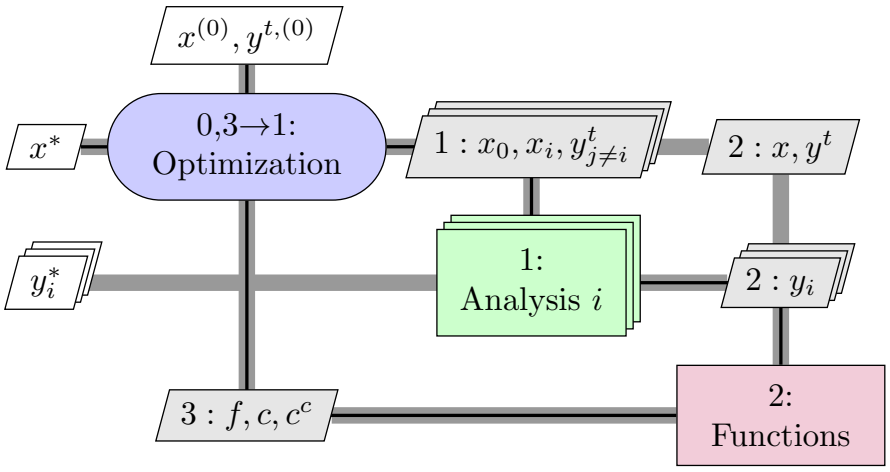


Figure 7.11: Diagram of the IDF architecture.

problem for the IDF architecture is

$$\begin{aligned}
\text{minimize} \quad & f_0\left(x, y\left(x, y^t\right)\right) \\
\text{with respect to} \quad & x, y^t \\
\text{subject to} \quad & c_0\left(x, y\left(x, y^t\right)\right) \geq 0 \\
& c_i\left(x_0, x_i, y_i\left(x_0, x_i, y^t_{j\neq i}\right)\right) \geq 0 \quad \text{for} \quad i = 1, \ldots, N \\
& c^c_i = y^t_i - y_i\left(x_0, x_i, y^t_{j\neq i}\right) = 0 \quad \text{for} \quad i = 1, \ldots, N.
\end{aligned} \tag{7.26}$$

The most important consequence of this reformulation is the removal of all state variables and discipline analysis equations from the problem statement. All coupling variables are now implicit functions of design variables and coupling variable targets as a result of solving the discipline analyses equations exactly at each optimization iteration.

The XDSM for IDF is shown in Fig. 7.11. This architecture enables the discipline analyses to be performed in parallel, since the coupling between the disciplines is resolved by the target coupling variables, y^t, and consistency constraints, c^c. Within the optimization iteration, specialized software for solving the discipline analyses can now be used to return coupling variable values to the objective and constraint function calculations. The net effect is that the IDF problem is both substantially smaller than the SAND problem and requires minimal modification to existing discipline analyses. In the field of PDE-constrained optimization, the IDF architecture is exactly analogous to a reduced-space method [16].

In spite of the reduced problem size when compared to SAND, the size of the IDF problem can still be an issue. If the number of coupling variables is large, the size of the resulting optimization problem might still be too large to solve efficiently. The large problem size can be mitigated to some extent by careful selection of the disciplinary variable partitions or aggregation of the coupling variables to reduce information transfer between disciplines.

If gradient-based optimization software is used to solve the IDF problem, gradient computation can become an issue for this architecture. When disciplinary analyses are expensive, evaluating the objective and constraint function gradients becomes a costly part of the optimization procedure. This is because the gradients themselves must be discipline-feasible, i.e., the changes in design

variables cannot cause the output coupling variables to violate the discipline analysis equations to first order.

In practice, gradients are often calculated using some type of finite-differencing procedure, where the discipline analysis is evaluated for each design variable. While this approach preserves disciplinary feasibility, it is costly and unreliable. If the discipline analysis code allows for the use of complex numbers, the complex-step method [110] is an alternative approach which gives machine-precision derivative estimates. If the analysis codes require a particularly long time to evaluate, the use of automatic differentiation or analytic derivative calculations (direct or adjoint methods) can be used to avoid multiple discipline analysis evaluations [106]. While the development time for these methods can be long, the reward is accurate derivative estimates and massive reductions in computational cost, especially for design optimization based on high-fidelity models.

Example 7.40. Aerostructural Optimization Using IDF

$$
\begin{aligned}
\text{minimize} \quad & -R && (7.27)\\
\text{w.r.t.} \quad & \Lambda, \gamma, t, \Gamma^t, \alpha^t, u^t && (7.28)\\
\text{s.t.} \quad & \sigma_{\text{yield}} - \sigma_i \geq 0 && (7.29)\\
& \Gamma^t - \Gamma = 0 && (7.30)\\
& \alpha^t - \alpha = 0 && (7.31)\\
& u^t - u = 0 && (7.32)
\end{aligned}
$$

7.5.4 Multidisciplinary Feasible (MDF)

If both analysis and consistency constraints are removed from Problem (7.18), we obtain the MDF architecture [37]. This architecture has also been referred to in the literature as Fully Integrated Optimization [4] and Nested Analysis and Design [10]. The resulting optimization problem is

$$
\begin{aligned}
\text{minimize} \quad & f_0\left(x, y\left(x, y\right)\right) \\
\text{with respect to} \quad & x \\
\text{subject to} \quad & c_0\left(x, y\left(x, y\right)\right) \geq 0 \\
& c_i\left(x_0, x_i, y_i\left(x_0, x_i, y_{j\neq i}\right)\right) \geq 0 \quad \text{for} \quad i = 1, \ldots, N.
\end{aligned}
\tag{7.33}
$$

The MDF architecture XDSM for three disciplines is shown in Fig. 7.12. Typically, a fixed point iteration, such as the block Gauss–Seidel iteration shown in Fig. 7.12, is used to converge the multidisciplinary analysis (MDA), where each discipline is solved in turn. This is usually an approach that exhibits slow convergence rates. Re-ordering the sequence of disciplines can improve the convergence rate of Gauss–Seidel [19], but even better convergence rates can be achieved through the use of Newton-based methods [75]. Due to the sequential nature of the Gauss–Seidel iteration, we cannot evaluate the disciplines in parallel and cannot apply our convention for compacting the XDSM. Using a different MDA method results in a different XDSM.

An obvious advantage of MDF over the other monolithic architectures is that the optimization problem is as small as it can be for a monolithic architecture, since only the design variables, objective function and design constraints are under the direct control of the optimizer. Another benefit is that MDF always returns a system design that always satisfies the consistency constraints, even if the optimization process is terminated early. This is advantageous in an engineering design context if time is limited and we are not as concerned with finding an optimal design in the strict mathematical sense as with finding an improved design. Note, however, that design constraint

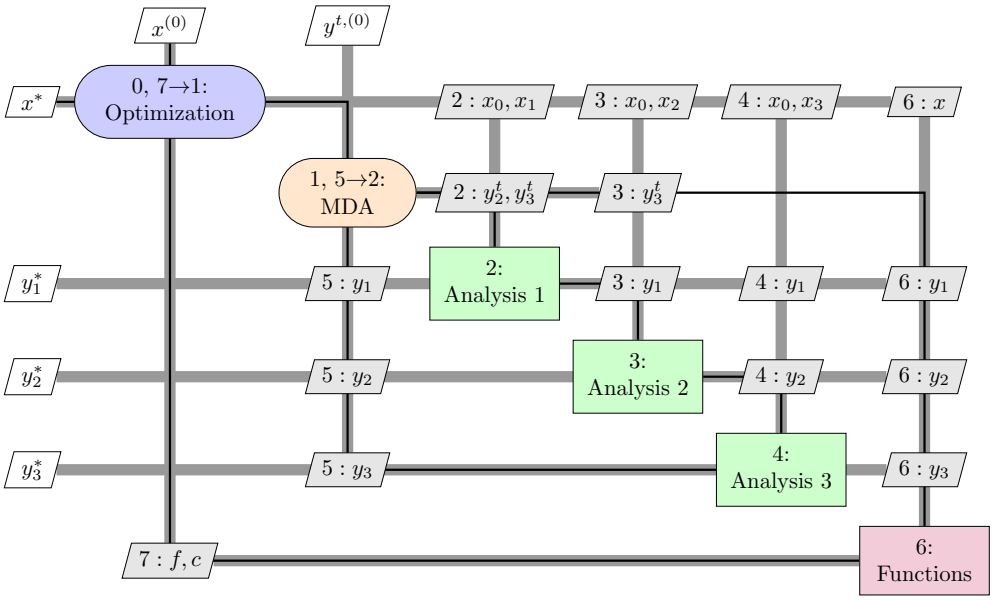


Figure 7.12: Diagram for the MDF architecture with a Gauss–Seidel multidisciplinary analysis.

satisfaction is not guaranteed if the optimization is terminated early; that depends on whether the optimization algorithm maintains a feasible design point or not. In particular, methods of feasible directions [168] require and maintain a feasible design point while many robust sequential quadratic programming [50] and interior point methods [169] do not.

The main disadvantage of MDF is that a consistent set of coupling variables must be computed and returned to the optimizer every time the objective and constraint functions are re-evaluated. In other words, the architecture requires a full MDA to be performed for every optimization iteration. Instead of simply running each individual discipline analysis once per optimization iteration, as we do in IDF, we need to run every discipline analysis multiple times until a consistent set of coupling variables is found. This task requires its own specialized iterative procedure outside of the optimization. Developing an MDA procedure can be time consuming if one is not already in place.

When using a gradient-based optimizer, gradient calculations are also much more difficult for MDF than for IDF. Just as the gradient information in IDF must be discipline-feasible, the gradient information under MDF must be feasible with respect to all disciplines. Fortunately, research in the sensitivity of coupled systems is fairly mature, and semi-analytic methods are available to drastically reduce the cost of this step by eliminating finite differencing over the full MDA [112, 150]. There is also some preliminary work towards automating the implementation of these coupled sensitivity methods [109]. The required partial derivatives can be obtained using any of the methods described in Section 7.5.3 for the individual disciplines in IDF.

Example 7.41. Aerostructural Optimization with MDF

$$\begin{aligned} \text{minimize} \quad & -R && (7.34)\\ \text{w.r.t.} \quad & \Lambda, \gamma, t && (7.35)\\ \text{s.t.} \quad & \sigma_{\text{yield}} - \sigma_i(u) \geq 0 && (7.36) \end{aligned}$$

Where the aerostructural analysis is as before:

$$A\Gamma - v(u, \alpha) = 0 \tag{7.37}$$
$$K(t, \Lambda)u - F(\Gamma) = 0 \tag{7.38}$$
$$L(\Gamma) - W(t) = 0 \tag{7.39}$$

7.6 Distributed Architectures

7.6.1 Motivation

Thus far, we have focused our discussion on monolithic MDO architectures: those that form and solve a single optimization problem. Many more architectures have been developed that decompose this single optimization problem into a set of smaller optimization problems, or subproblems, that have the same solution when reassembled. These are the *distributed* MDO architectures. Before reviewing and classifying the distributed architectures, we discuss the motivation of MDO researchers in developing this new class of MDO architectures.

Early in the history of optimization, the motivation for decomposition methods was to exploit the structure of the problem to reduce solution time. Many large optimization problems, such as network flow problems and resource allocation problems, exhibit such special structures [95].

To better understand decomposition, consider the following problem:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{N} f_i(x_i) \\ \text{with respect to} \quad & x_1, \ldots, x_N \\ \text{subject to} \quad & c_0(x_1, \ldots, x_N) \geq 0 \\ & c_1(x_1) \geq 0, \ldots, c_N(x_N) \geq 0. \end{aligned} \tag{7.40}$$

In this problem, there are no shared design variables, x_0, and the objective function is separable, i.e. it can be expressed as a sum of functions, each of which depend only on the corresponding local design variables, x_i. On the other hand, the constraints include a set of constraints, c_0, that depends on more than one set of design variables. This problem is referred to as a *complicating constraints problem* [35]; if c_0 did not exist, we could simply decompose this optimization problem into N independent problems.

Another possibility is that a problem includes shared design variables and a separable objective function, with no complicating constraints, i.e.,

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{N} f_i(x_0, x_i) \\ \text{with respect to} \quad & x_0, x_1, \ldots, x_N \\ \text{subject to} \quad & c_1(x_0, x_1) \geq 0, \ldots, c_N(x_0, x_N) \geq 0. \end{aligned} \tag{7.41}$$

This is referred to as a problem with *complicating variables* [35]. In this case, the decomposition would be straightforward if there were no shared design variables, x_0, and we could solve N optimization problems independently and in parallel.

Specialized decomposition methods were developed to reintroduce the complicating variables or constraints into these problems with only small increases in time and cost relative to the N independent problems. Examples of these methods include Dantzig–Wolfe decomposition [38] and

Benders decomposition [12] for Problems (7.40) and (7.41), respectively. However, these decomposition methods were designed to work with the simplex algorithm on linear programming problems. In the simplex algorithm, the active set changes only by one constraint at a time so decomposition is the only way to exploit the special problem structure. However, algorithms for nonlinear optimization that are based on Newton's method, such as sequential quadratic programming and interior point methods, may also use specialized matrix factorization techniques to exploit sparsity structures in the problem. While nonlinear decomposition algorithms were later developed, to the best of our knowledge, no performance comparisons have been made between these decomposition algorithms and Newton-like algorithms employing sparse matrix factorization. Intuition suggests that the latter should be faster due to the ability of Newton methods to exploit second-order problem information. Thus, while decomposition methods do exist for nonlinear problems, problem structure is not the primary motivation for their development.

The primary motivation for decomposing the MDO problem comes from the structure of the engineering design environment. Typical industrial practice involves breaking up the design of a large system and distributing aspects of that design to specific engineering groups. These groups may be geographically distributed and may only communicate infrequently. More importantly, however, these groups typically like to retain control of their own design procedures and make use of in-house expertise, rather than simply passing on discipline analysis results to a central design authority [92]. Decomposition through distributed architectures allow individual design groups to work in isolation, controlling their own sets of design variables, while periodically updating information from other groups to improve their aspect of the overall design. This approach to solving the problem conforms more closely with current industrial design practice than the approach of the monolithic architectures.

In an industrial setting, the notion of a "discipline" in an MDO problem can take on many forms. Traditionally, disciplines have been defined in terms of knowledge areas — aerodynamics, structures, propulsion, and control are examples of aircraft design disciplines. This definition conforms well with many existing analysis codes. In principle, however, a discipline can constitute any suitable partition of an MDO problem. For example, if the structural design is broken down by component, (e.g. wings, tail, and fuselage), the design of each component could also be considered a discipline in an MDO problem. Therefore, an industrial-scale MDO problem could contain hundreds of disciplines, depending on the company architecture. How the disciplines are arranged within a distributed MDO architecture is up to the company, but there is some literature for choosing disciplinary partitions to reduce coupling between distributed groups [120, 121**?**].

The structure of disciplinary design groups working in isolation has a profound effect on the timing of each discipline analysis evaluation. In a monolithic architecture, all discipline analysis programs are run exactly the same number of times, based on requests from the optimizer or MDA program. In the context of parallel computing, this approach can be thought of as a synchronous algorithm [15]. In instances where some analyses or optimizations are much more expensive than others, such as the case of multifidelity optimization [135, 172], the performance suffers because the processors performing the inexpensive analyses and optimizations experience long periods of inactivity while waiting to update their available information. In the language of parallel computing, the computation is said to exhibit poor load balancing. Another example of this case is aerostructural optimization, in which a nonlinear aerodynamics solver may require an order of magnitude more time to run than a linear structural solver [33]. By decomposing the optimization problem, the processor work loads may be balanced by allowing disciplinary analyses with lower computational cost to perform more optimization on their own. Those disciplines with less demanding optimizations may also be allowed to make more progress before updating nonlocal information. In other words, the whole design process occurs not only in parallel but also asynchronously. While the

asynchronous design process may result in more total computational effort, the intrinsically parallel nature of architecture allows much of the work to proceed concurrently, reducing the wall-clock time of the optimization.

7.6.2 Classification

We now introduce a new approach to classifying MDO architectures. Some of the previous classifications of MDO architectures were based on observations of which constraints were available to the optimizer to control [3, 10]. Alexandrov and Lewis [3] used the term "closed" to denote when a set of constraints cannot be satisfied by explicit action of the optimizer, and "open" otherwise. For example, the MDF architecture is closed with respect to both analysis and consistency constraints, because their satisfaction is determined through the process of converging the MDA. Similarly, IDF is closed analysis but open consistency since the consistency constraints can be satisfied by the optimizer adjusting the coupling targets and design variables. Tosserams et al. [163] expanded on this classification scheme by discussing whether or not distributed architectures used open or closed local design constraints in the system subproblem. Closure of the constraints is an important consideration when selecting an architecture, because most robust optimization software will permit the exploration of infeasible regions of the design space. Such exploration can result in faster solutions via fewer optimization iterations, but this must be weighed against the increased optimization problem size and the risk of terminating the optimization at an infeasible point.

The central idea in our classification is that distributed MDO architectures can be classified based on their monolithic analogues: either MDF, IDF, or SAND. This stems from the different approaches to handling the state and coupling variables in the monolithic architectures. Those distributed architectures that follow MDF and use a multidisciplinary analysis (or an approximation of an MDA) to enforce coupling variable consistency at the final solution are classified as distributed MDF architectures. Similarly, those distributed architectures that follow IDF and use coupling variable targets and consistency constraints to enforce consistency at the final solution are classified as distributed IDF architectures.

Our classification is similar to the previous classifications in that an equality constraint must be removed from the optimization problem — i.e., closed — for every variable removed from the problem statement. However, using a classification based on the monolithic architectures makes it much easier to see the connections between distributed architectures, even when these architectures are developed in isolation from each other. In many cases, the problem formulations in the distributed architecture can be derived directly from that of the monolithic architecture by adding certain elements to the problem, by making certain assumptions, and by applying a specific decomposition scheme. Our classification can also be viewed as a framework in which researchers could develop new distributed architectures, since the starting point for a distributed architecture is *always* a monolithic architecture.

This classification of architectures is represented in Fig. 7.13. Known relationships between the architectures are shown by arrows. Due to the large number of adaptations created for some distributed architectures, such as the introduction of surrogate models and variations to solve multiobjective problems, we have only included the "core" architectures in our diagram. Details on the available variations for each distributed architecture are presented in the relevant sections.

None of the distributed architectures developed to date have been considered analogues of SAND. As discussed in Section 7.5, the desire to use independent "black-box" computer codes for the disciplinary analyses necessarily excluded consideration of the SAND problem formulation as a starting point. Nevertheless, the techniques used to derive distributed architectures from IDF and MDF may also be useful when using SAND as a foundation.

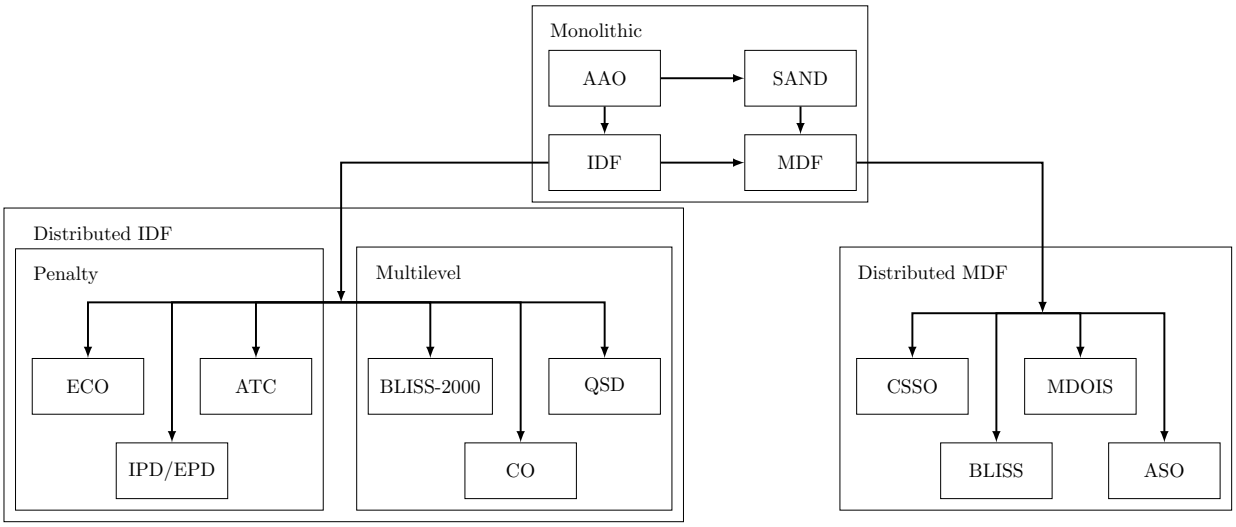


Figure 7.13: Classification of the MDO architectures.

Our classification scheme does not distinguish between the different solution techniques for the distributed optimization problems. For example, we have not focused on the order in which the distributed problems are solved. Coordination schemes are partially addressed in the Distributed IDF group, where we have classified the architectures as either "penalty" or "multilevel", based on whether penalty functions or a problem hierarchy is used in the coordination. This grouping follows from the work of de Wit and van Keulen [40].

One area that is not well-explored in MDO is the use of hybrid architectures. By hybrid, we mean an architecture that incorporates elements of two or more other architectures in such a way that different disciplinary analyses or optimizations are treated differently. For example, a hybrid monolithic architecture could be created from MDF and IDF by resolving the coupling of some disciplines within an MDA, while the remaining coupling variables are resolved through constraints. Some ideas for hybrid architectures have been proposed by Marriage and Martins [109] and Geethaikrishnan et al. [48]. Such architectures could be especially useful in specific applications where the coupling characteristics vary widely among the disciplines. However, general rules need to be developed to say under what conditions the use of certain architectures is advantageous. As we note in Section 7.7, much work remains in this area.

In the following sections, in chronological order of their initial development, we introduce the distributed architectures for MDO. We prefer to use the term "distributed" as opposed to "hierarchical" or "multilevel", because these architectures do not necessarily create a hierarchy of problems to solve. In some cases, it is better to think of all optimization problems as being on the same level. Furthermore, neither the systems being designed nor the design team organization need to be hierarchical in nature for these architectures to be applicable. Our focus here is to provide a unified description of these architectures and explain some advantages and disadvantages of each. Along the way, we will point out variations and applications of each architecture that can be found in the literature. We also aim to review the state-of-the-art in architectures, since the most recent detailed architecture survey in the literature dates back from more than a decade ago [148]. More recent surveys, such as that of Agte et al. [1], discuss MDO more generally without detailing the architectures themselves.

7.6.3 Concurrent Subspace Optimization (CSSO)

CSSO is one of the oldest distributed architectures for large-scale MDO problems. The original formulation [18, 149] decomposes the system problem into independent subproblems with disjoint sets of variables. Global sensitivity information is calculated at each iteration to give each subproblem a linear approximation to a multidisciplinary analysis, improving the convergence behavior. At the system level, a coordination problem is solved to recompute the "responsibility", "tradeoff", and "switch" coefficients assigned to each discipline to provide information on design variable preferences for nonlocal constraint satisfaction. Using these coefficients gives each discipline a certain degree of autonomy within the system as a whole. Shankar et al. [144] proposed several improvements to the original architecture, including methods for updating the coefficients, and tested them on two- and three-variable quadratic optimization problems. Unfortunately, they note that the architecture performance is sensitive to parameter selection and extensive tuning may be required to run CSSO efficiently on larger nonlinear problems.

Several variations of this architecture have been developed to incorporate surrogate models [134, 143, 170] and higher-order information sharing among the disciplines [133]. More recently, the architecture has also been adapted to solve multiobjective problems [68, 126, 174]. Parashar and Bloebaum [127] extended a multiobjective CSSO formulation to handle robust design optimization problems. An application of the architecture to the design of high-temperature aircraft engine components is presented by Tappeta et al. [158].

The version we consider here, due to Sellar et al. [143], uses surrogate models of each disciplinary analysis to efficiently model multidisciplinary interactions. Using our unified notation, the CSSO system subproblem is given by

$$\begin{aligned} \text{minimize} \quad & f_0\left(x, \tilde{y}\left(x, \tilde{y}\right)\right) \\ \text{with respect to} \quad & x \\ \text{subject to} \quad & c_0\left(x, \tilde{y}\left(x, \tilde{y}\right)\right) \geq 0 \\ & c_i\left(x_0, x_i, \tilde{y}_i\left(x_0, x_i, \tilde{y}_{j\neq i}\right)\right) \geq 0 \,\text{for} \quad i = 1, \ldots, N \end{aligned} \tag{7.42}$$

and the discipline i subproblem is given by

$$\begin{aligned} \text{minimize} \quad & f_0\left(x, y_i\left(x_i, \tilde{y}_{j\neq i}\right), \tilde{y}_{j\neq i}\right) \\ \text{with respect to} \quad & x_0, x_i \\ \text{subject to} \quad & c_0\left(x, \tilde{y}\left(x, \tilde{y}\right)\right) \geq 0 \\ & c_i\left(x_0, x_i, y_i\left(x_0, x_i, \tilde{y}_{j\neq i}\right)\right) \geq 0 \\ & c_j\left(x_0, \tilde{y}_j\left(x_0, \tilde{y}\right)\right) \geq 0 \quad \text{for} \quad j = 1, \ldots, N \, j \neq i. \end{aligned} \tag{7.43}$$

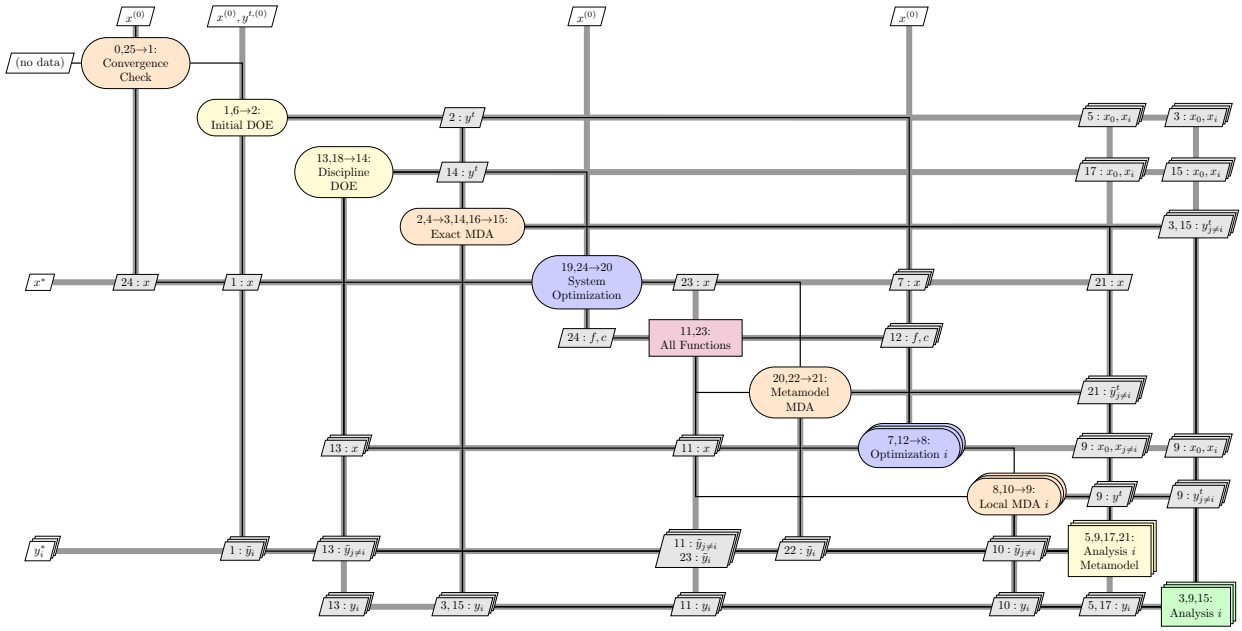


Figure 7.14: Diagram for the CSSO architecture.

The CSSO architecture is depicted in Fig. 7.14 and the corresponding steps are listed in Algorithm 16. A potential pitfall of this architecture is the necessity of including all design variables in the system subproblem. For industrial-scale design problems, this may not always be possible or practical.

Algorithm 16 CSSO

Input: Initial design variables x
Output: Optimal variables x^*, objective function f^*, and constraint values c^*
0: Initiate main CSSO iteration
repeat
1: Initiate a design of experiments (DOE) to generate design points
for Each DOE point **do**
2: Initiate an MDA that uses exact disciplinary information
repeat
3: Evaluate discipline analyses
4: Update coupling variables y
until $4 \rightarrow 3$: MDA has converged
5: Update the disciplinary surrogate models with the latest design
end for $6 \rightarrow 2$
7: Initiate independent disciplinary optimizations (in parallel)
for Each discipline i **do**
repeat
8: Initiate an MDA with exact coupling variables for discipline i and approximate coupling variables for the other disciplines
repeat
9: Evaluate discipline i outputs y_i, and surrogate models for the other disciplines, $\tilde{y}_{j \neq i}$
until $10 \rightarrow 9$: MDA has converged
11: Compute objective f_0 and constraint functions c using current data
until $12 \rightarrow 8$: Disciplinary optimization i has converged
end for
13: Initiate a DOE that uses the subproblem solutions as sample points
for Each subproblem solution i **do**
14: Initiate an MDA that uses exact disciplinary information
repeat
15: Evaluate discipline analyses.
until $16 \rightarrow 15$ MDA has converged
17: Update the disciplinary surrogate models with the newest design
end for $18 \rightarrow 14$
19: Initiate system-level optimization
repeat
20: Initiate an MDA that uses only surrogate model information
repeat
21: Evaluate disciplinary surrogate models
until $22 \rightarrow 21$: MDA has converged
23: Compute objective f_0, and constraint function values c
until $24 \rightarrow 20$: System level problem has converged
until $25 \rightarrow 1$: CSSO has converged

There have been some benchmarks comparing CSSO with other MDO architectures. Perez et al., [129] Yi et al. [171], and Tedford and Martins [159] all compare CSSO to other architectures on low-dimensional test problems with gradient-based optimization. Their results all show that CSSO required many more analysis calls and function evaluations than other architectures to converge to

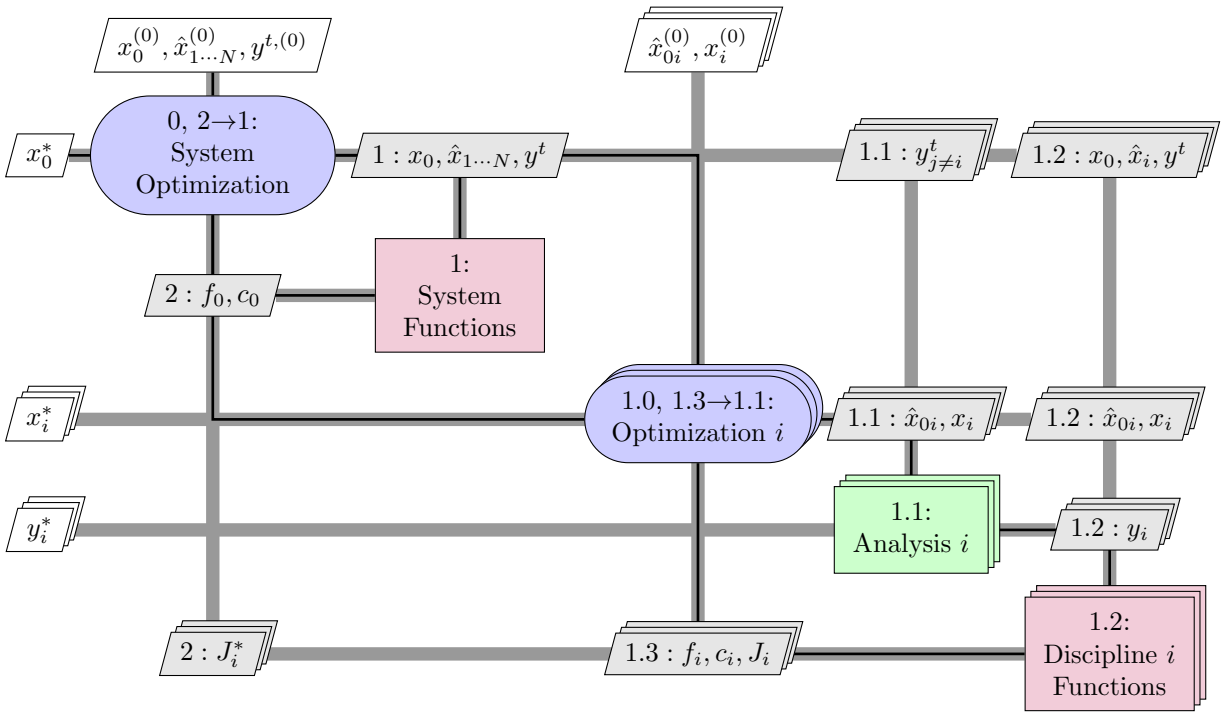


Figure 7.15: Diagram for the CO architecture.

an optimal design. The results of de Wit and van Keulen [39] showed that CSSO was unable to reach the optimal solution of even a simple minimum-weight two-bar truss problem. Thus, CSSO seems to be largely ineffective when compared with newer MDO architectures.

7.6.4 Collaborative Optimization (CO)

In CO, the disciplinary optimization problems are formulated to be independent of each other by using target values of the coupling and shared design variables [23, 25]. These target values are then shared with all disciplines during every iteration of the solution procedure. The complete independence of disciplinary subproblems combined with the simplicity of the data-sharing protocol makes this architecture attractive for problems with a small amount of shared design information.

Braun [23] formulated two versions of the CO architecture: CO_1 and CO_2. CO_2 is the most frequently used of these two original formulations, so it will be the focus of our discussion. The CO_2 system subproblem is given by:

$$\begin{aligned} \text{minimize} \quad & f_0\left(x_0, \hat{x}_1, \ldots, \hat{x}_N, y^t\right) \\ \text{with respect to} \quad & x_0, \hat{x}_1, \ldots, \hat{x}_N, y^t \\ \text{subject to} \quad & c_0\left(x_0, \hat{x}_1, \ldots, \hat{x}_N, y^t\right) \geq 0 \\ & J_i^* = ||\hat{x}_{0i} - x_0||_2^2 + ||\hat{x}_i - x_i||_2^2 + \\ & \qquad ||y_i^t - y_i\left(\hat{x}_{0i}, x_i, y_{j\neq i}^t\right)||_2^2 = 0 \quad \text{for} \quad i = 1, \ldots, N \end{aligned} \tag{7.44}$$

where $\hat{x}_{0i}$ are duplicates of the global design variables passed to (and manipulated by) discipline i and $\hat{x}_i$ are duplicates of the local design variables passed to the system subproblem. Duplicates of

the local design variables are made only if those variables directly influence the objective. In CO_1, the quadratic equality constraints are replaced with linear equality constraints for each target-response pair. In either case, if derivatives are required to solve the system subproblem, they must be computed with respect to the *optimized* function J_i^*. Even though the architecture has yet to converge on an optimized system design, this step is still referred to as a post-optimality analysis because the subsystems have been optimized with respect to their local information.

The discipline i subproblem in both CO_1 and CO_2 is

$$\begin{aligned} \text{minimize} \quad & J_i\left(\hat{x}_{0i}, x_i, y_i\left(\hat{x}_{0i}, x_i, y_{j\neq i}^t\right)\right) \\ \text{with respect to} \quad & \hat{x}_{0i}, x_i \\ \text{subject to} \quad & c_i\left(\hat{x}_{0i}, x_i, y_i\left(\hat{x}_{0i}, x_i, y_{j\neq i}^t\right)\right) \geq 0. \end{aligned} \tag{7.45}$$

Thus the system-level problem is responsible for minimizing the design objective, while the discipline level problems minimize system inconsistency. Braun [23] showed that the CO problem statement is mathematically equivalent to the IDF problem statement (7.26) and, therefore, equivalent to the original MDO problem (7.18) as well. In particular, if the CO architecture converges to a point that locally minimizes f_0 while satisfying the design constraints c_0 and c_i and consistency constraints $J_i = 0$, the resulting point must also be a local minimum of the IDF problem. This idea can be inferred by the special structure of problems (7.44) and (7.45). CO is depicted by the XDSM in Fig. 7.15. The corresponding procedure is detailed in Algorithm 17.

Algorithm 17 Collaborative optimization

Input: Initial design variables x
Output: Optimal variables x^*, objective function f^*, and constraint values c^*
0: Initiate system optimization iteration
repeat
1: Compute system subproblem objectives and constraints
for Each discipline i (in parallel) **do**
1.0: Initiate disciplinary subproblem optimization
repeat
1.1: Evaluate disciplinary analysis
1.2: Compute disciplinary subproblem objective and constraints
1.3: Compute new disciplinary subproblem design point and J_i
until 1.3 $\rightarrow$ 1.1: Optimization i has converged
end for
2: Compute a new system subproblem design point
until 2 $\rightarrow$ 1: System optimization has converged

In spite of the organizational advantage of having fully separate disciplinary subproblems, CO has major weaknesses in the mathematical formulation that lead to poor performance in practice [4, 42]. In particular, the system problem in CO_1 has more equality constraints than variables, so if the system cannot be made fully consistent, the system subproblem is infeasible. This can also happen in CO_2, but it is not the most problematic issue. The most significant difficulty with CO_2 is that the constraint gradients of the system problem at an optimal solution are all zero vectors. This represents a breakdown in the constraint qualification of the Karush–Kuhn–Tucker optimality conditions, which slows down convergence for most gradient-based optimization software [4]. In the worst case, the CO_2 formulation may not converge at all. These difficulties with the original formulations of CO have inspired several researchers to develop modifications to improve the behavior of

the architecture. In a few cases, problems have been solved with CO and a gradient-free optimizer, such as a genetic algorithm [173], or a gradient-based optimizer that does not use the Lagrange multipliers in the termination condition [101] to handle the troublesome constraints. While such approaches do avoid the obvious problems with CO, they bring other issues. Gradient-free optimizers that do not employ some kind of surrogate modeling tend to require far more function evaluations than gradient-based optimizers. These additional function evaluations and disciplinary analyses can become a bottleneck within the CO architecture computations. Gradient-based optimizers that do not terminate based on Lagrange multiplier values, such as feasible direction methods, often fail in nonconvex feasible regions. As pointed out by DeMiguel [42], the CO system subproblem is set-constrained, i.e., nonconvex, because of the need to satisfy optimality in the disciplinary subproblems.

To run CO successfully with robust gradient-based optimization methods, several ideas have surfaced in the literature. The approach taken by DeMiguel and Murray [42] to fix the problems with CO is to relax the troublesome constraints using an L_1 exact penalty function with a fixed penalty parameter value and add elastic variables to preserve the smoothness of the problem. This revised approach is called Modified Collaborative Optimization (MCO). This approach satisfies the requirement of mathematical rigor, as algorithms using the penalty function formulation are known to converge to an optimal solution under mild assumptions [46, 123]. However, the test results of Brown and Olds [26] show strange behavior in a practical design problem. In particular, they observed that for values of the penalty parameter above a threshold value, the problem could not improve the initial design point. Below a lower threshold value, the architecture showed very poor convergence. Finally, a penalty parameter could not be found that produced a final design close to those computed by other architectures. In light of these findings, the authors rejected MCO from further testing.

Another idea, proposed by Sobieski and Kroo [147], uses surrogate models to approximate the post-optimality behavior of the disciplinary subproblems in the system subproblem. This both eliminates the direct calculation of post-optimality derivatives and improves the treatment of the consistency constraints. While the approach does seem to be effective for the problems they solve, to our knowledge, it has not been adopted by any other researchers to date.

The simplest and most effective known fix for the difficulties of CO involves relaxing the system subproblem equality constraints to inequalities with a relaxation tolerance, which was originally proposed by Braun et al. [25]. This approach was also successful in other test problems [108, 128], where the choice of tolerance is a small fixed number, usually 10^{-6}. The effectiveness of this approach stems from the fact that a positive inconsistency value causes the gradient of the constraint to be nonzero if the constraint is active, eliminating the constraint qualification issue. Nonzero inconsistency is not an issue in a practical design setting provided the inconsistency is small enough such that other errors in the computational model dominate at the final solution. Li et al. [98] build on this approach by adaptively choosing the tolerance during the solution procedure so that the system-level problem remains feasible at each iteration. This approach appears to work when applied to the test problems in [4], but has yet to be verified on larger test problems.

Despite the numerical issues, CO has been widely implemented on a number of MDO problems. Most of applications are in the design of aerospace systems. Examples include the design of launch vehicles [22], rocket engines [29], satellite constellations [28], flight trajectories [24, 96], flight control systems [130], preliminary design of complete aircraft [93, 107], and aircraft family design [6]. Outside aerospace engineering, CO has been applied to problems involving automobile engines [116], bridge design [9], railway cars [44], and even the design of a scanning optical microscope [132].

Adaptations of the CO architecture have also been developed for multiobjective, robust, and multifidelity MDO problems. Multiobjective formulations of CO were first described by Tappeta

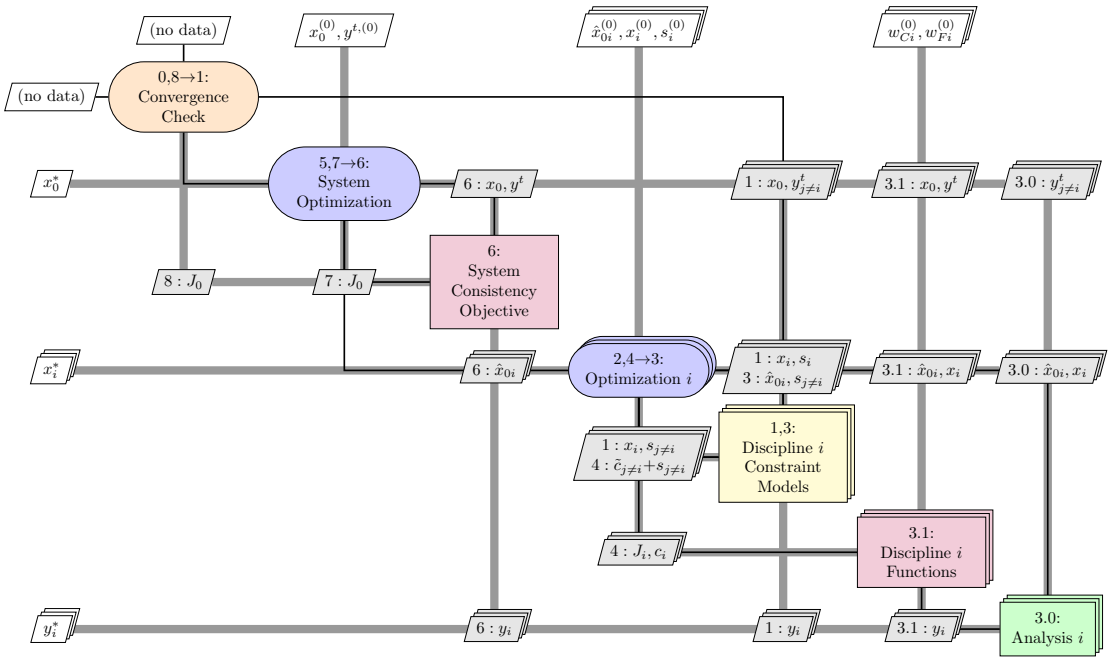


Figure 7.16: XDSM for the ECO architecture

and Renaud [157]. McAllister et al. [115] present a multiobjective approach using linear physical programming. Available robust design formulations incorporate the decision-based models of Gu et al. [55] and McAllister and Simpson [116], the implicit uncertainty propagation method of Gu et al. [56], and the fuzzy computing models of Huang et al. [71]. Multiple model fidelities were integrated into CO for an aircraft design problem by Zadeh and Toropov [172].

The most recent version of CO — Enhanced Collaborative Optimization (ECO) — was developed by Roth and Kroo [136, 137]. Fig. 7.16 shows the XDSM corresponding to this architecture. The problem formulation of ECO, while still being derived from the same basic problem as the original CO architecture, is radically different and therefore deserves additional attention. In a sense, the roles of the system and discipline optimization have been reversed in ECO when compared to CO. In ECO the system subproblem minimizes system infeasibility, while the disciplinary subproblems minimize the system objective. The system subproblem is

$$\begin{aligned} \text{minimize} \quad & J_0 = \sum_{i=1}^{N} ||\hat{x}_{0i} - x_0||_2^2 + ||y_i^t - y_i\left(x_0, x_i, y_{j\neq i}^t\right)||_2^2 \\ \text{with respect to} \quad & x_0, y^t. \end{aligned} \tag{7.46}$$

Note that this subproblem is unconstrained. Also, unlike CO, post-optimality derivatives are not required by the system subproblem, because the disciplinary responses are treated as parameters. The system subproblem chooses the shared design variables by averaging all disciplinary preferences.

The i^{th} disciplinary subproblem is

$$\begin{aligned}
\text{minimize} \quad & J_i = \tilde{f}_0\left(\hat{x}_{0i}, y_i\left(\hat{x}_{0i}, x_i, y^t_{j\neq i}\right)\right) + \\
& w_{Ci}\left(||\hat{x}_{0i} - x_0||_2^2 + ||y_i^t - y_i\left(\hat{x}_{0i}, x_i, y^t_{j\neq i}\right)||_2^2\right) + \\
& w_{Fi}\sum_{j=1, j\neq i}^{N}\sum_{k=1}^{n_s} s_{jk} \\
\text{with respect to} \quad & \hat{x}_{0i}, x_i, s_{j\neq i} \\
\text{subject to} \quad & c_i\left(\hat{x}_{0i}, x_i, y_i\left(\hat{x}_{0i}, x_i, y^t_{j\neq i}\right)\right) \geq 0 \\
& \tilde{c}_{j\neq i}\left(\hat{x}_{0i}\right) - s_{j\neq i} \geq 0 \qquad j = 1, \ldots, N \\
& s_{j\neq i} \geq 0 \qquad j = 1, \ldots, N,
\end{aligned} \tag{7.47}$$

where w_{Ci} and w_{Fi} are penalty weights for the consistency and nonlocal design constraints, and s is a local set of elastic variables for the constraint models. The w_{Fi} penalty weights are chosen to be larger than the largest Lagrange multiplier, while the w_{Ci} weights are chosen to guide the optimization toward a consistent solution. Theoretically, each w_{Ci} must be driven to infinity to enforce consistency exactly. However, smaller finite values are used in practice to both provide an acceptable level of consistency and explore infeasible regions of the design space [136].

The main new idea introduced in ECO is to include linear models of nonlocal constraints, represented by $\tilde{c}_{j\neq i}$, and a quadratic model of the system objective function in each disciplinary subproblem, represented by $\tilde{f}_0$. This is meant to increase each discipline's "awareness" of their influence on other disciplines and the global objective as a whole. The construction of the constraint models deserves special attention, because it strongly affects the structure of Fig. 7.16. The constraint models for each discipline are constructed by first solving the optimization problem that minimizes the constraint violation with respect to local elastic and design variables, i.e.,

$$\begin{aligned}
\text{minimize} \quad & \sum_{k=1}^{n_s} s_{ik} \\
\text{with respect to} \quad & x_i, s_i \\
\text{subject to} \quad & c_i\left(x_0, x_i, y_i\left(x_0, x_i, y^t_{j\neq i}\right)\right) + s_i \geq 0 \\
& s_i \geq 0,
\end{aligned} \tag{7.48}$$

where the shared design variables and the coupling targets are treated as fixed parameters. Post-optimality derivatives are then computed to determine the change in the optimized local design variables with respect to the change in shared design variables. Combining these post-optimality derivatives with the appropriate partial derivatives yields the linear constraint models. The optimized local design variables and elastic variables from Problem (7.48) are then used as part of the initial data for Problem (7.47). The full algorithm for ECO is listed in Algorithm 18.

Algorithm 18 Enhanced collaborative optimization

Input: Initial design variables x
Output: Optimal variables x^*, objective function f^*, and constraint values c^*
0: Initiate ECO iteration
repeat
  for Each discipline i **do**
    1: Create linear constraint model
    2: Initiate disciplinary subproblem optimization
    repeat
      3: Evaluate nonlocal constraint models with local duplicates of shared variables
      3.0: Evaluate disciplinary analysis
      3.1: Compute disciplinary subproblem objective and constraints
      4: Compute new disciplinary subproblem design point and J_i
    until $4 \rightarrow 3$: Disciplinary optimization subproblem has converged
  end for
  5: Initiate system optimization
  repeat
    6: Compute J_0
    7: Compute updated values of x_0 and y^t.
  until $7 \rightarrow 6$: System optimization has converged
until $8 \rightarrow 1$: The J_0 is below specified tolerance

Based on Roth's results [136, 137], ECO is effective in reducing the number of discipline analyses compared to CO. The trade-off is in the additional time required to build and update the models for each discipline, weighed against the simplified solution to the decomposed optimization problems. The results also show that ECO compares favorably with the Analytical Target Cascading architecture, which we describe in Section 7.6.6.

While ECO seems to be effective, CO tends to be an inefficient architecture for solving MDO problems. Without any of the modifications discussed in this section, the architecture always requires a disproportionately large number of function and discipline evaluations [39, 84, 87, 171], assuming it converges at all. When the system-level equality constraints are relaxed, the results from CO are more competitive with other distributed architectures [108, 129, 159] but still compare poorly with the results of monolithic architectures.

Example 7.42. Aerostructural Optimization Using CO

System-level problem:

$$\begin{aligned} \text{minimize} \quad & -R && (7.49)\\ \text{w.r.t.} \quad & \Lambda^t, \Gamma^t, \alpha^t, u^t, W^t && (7.50)\\ \text{s.t.} \quad & J_1^* \leq 10^{-6} && (7.51)\\ & J_2^* \leq 10^{-6} && (7.52) \end{aligned}$$

Aerodynamics subproblem:

$$\begin{aligned} \text{minimize} \quad & J_1 = \left(1 - \frac{\Lambda}{\Lambda^t}\right)^2 + \sum \left(1 - \frac{\Gamma_i}{\Gamma_i^t}\right)^2 + \left(1 - \frac{\alpha}{\alpha^t}\right)^2 + \left(1 - \frac{W}{W^t}\right)^2 && (7.53)\\ \text{w.r.t.} \quad & \Lambda, \gamma, \alpha && (7.54)\\ \text{s.t.} \quad & L - W = 0 && (7.55) \end{aligned}$$

Structures subproblem:

$$\begin{aligned} \text{minimize} \quad & J_2 = \left(1 - \frac{\Lambda}{\Lambda^t}\right)^2 + \sum \left(1 - \frac{u_i}{u_i^t}\right)^2 && (7.56)\\ \text{w.r.t.} \quad & \Lambda, t && (7.57)\\ \text{s.t.} \quad & \sigma_{\text{yield}} - \sigma_i \geq 0 && (7.58) \end{aligned}$$

7.6.5 Bilevel Integrated System Synthesis (BLISS)

The BLISS architecture [153], like CSSO, is a method for decomposing the MDF problem along disciplinary lines. Unlike CSSO, however, BLISS assigns local design variables to disciplinary subproblems and shared design variables to the system subproblem. The basic approach of the architecture is to form a path in the design space using a series of linear approximations to the original design problem, with user-defined bounds on the design variable steps, to prevent the design point from moving so far away that the approximations are too inaccurate. This is an idea similar to that of trust-region methods [36]. These approximations are constructed at each iteration using global sensitivity information. The system level subproblem is formulated as

$$\begin{aligned} \text{minimize} \quad & (f_0^*)_0 + \left(\frac{\mathrm{d}f_0^*}{\mathrm{d}x_0}\right)\Delta x_0 \\ \text{with respect to} \quad & \Delta x_0 \\ \text{subject to} \quad & (c_0^*)_0 + \left(\frac{\mathrm{d}c_0^*}{\mathrm{d}x_0}\right)\Delta x_0 \geq 0 \\ & (c_i^*)_0 + \left(\frac{\mathrm{d}c_i^*}{\mathrm{d}x_0}\right)\Delta x_0 \geq 0 \quad \text{for} \quad i = 1, \ldots, N \\ & \Delta x_{0L} \leq \Delta x_0 \leq \Delta x_{0U}. \end{aligned} \tag{7.59}$$

The discipline i subproblem is given by

$$\begin{aligned} \text{minimize} \quad & (f_0)_0 + \left(\frac{\mathrm{d}f_0}{\mathrm{d}x_i}\right)\Delta x_i \\ \text{with respect to} \quad & \Delta x_i \\ \text{subject to} \quad & (c_0)_0 + \left(\frac{\mathrm{d}c_0}{\mathrm{d}x_i}\right)\Delta x_i \geq 0 \\ & (c_i)_0 + \left(\frac{\mathrm{d}c_i}{\mathrm{d}x_i}\right)\Delta x_i \geq 0 \\ & \Delta x_{iL} \leq \Delta x_i \leq \Delta x_{iU}. \end{aligned} \tag{7.60}$$

Note the extra set of constraints in both system and discipline subproblems denoting the design variables bounds.

In order to prevent violation of the disciplinary constraints by changes in the shared design variables, post-optimality derivative information (the change in the optimized disciplinary constraints with respect to a change in the system design variables) is required to solve the system subproblem. For this step, Sobieski [153] presents two methods: one based on a generalized version of the Global Sensitivity Equations [150], and another based on the "pricing" interpretation of local Lagrange multipliers. The resulting variants of BLISS are BLISS/A and BLISS/B, respectively. Other variations use surrogate model approximations to compute post-optimality derivative data [80, 85].

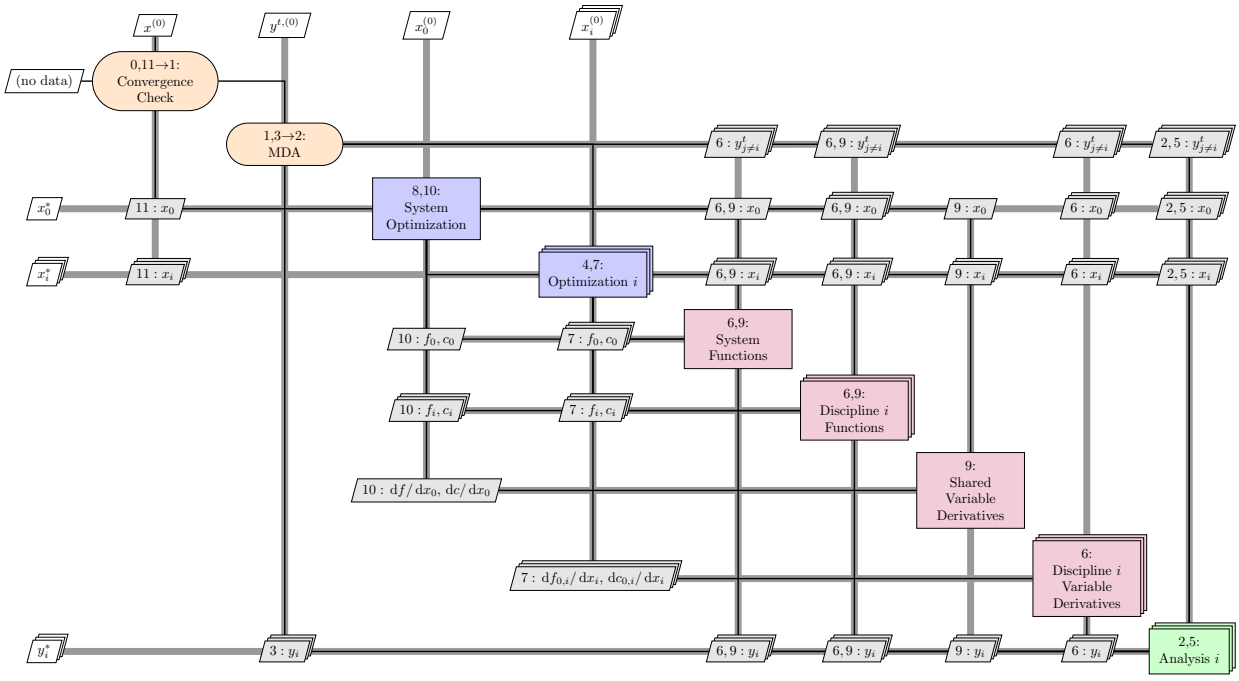


Figure 7.17: Diagram for the BLISS architecture

Algorithm 19 BLISS

Input: Initial design variables x
Output: Optimal variables x^*, objective function f^*, and constraint values c^*
0: Initiate system optimization
repeat
1: Initiate MDA
repeat
2: Evaluate discipline analyses
3: Update coupling variables
until 3 → 2: MDA has converged
4: Initiate parallel discipline optimizations
for Each discipline i **do**
5: Evaluate discipline analysis
6: Compute objective and constraint function values and derivatives with respect to local design variables
7: Compute the optimal solutions for the disciplinary subproblem
end for
8: Initiate system optimization
9: Compute objective and constraint function values and derivatives with respect to shared design variables using post-optimality analysis
10: Compute optimal solution to system subproblem
until 11 → 1: System optimization has converged

Fig. 7.17 shows the XDSM for BLISS and the procedure is listed in Algorithm 19. Due to

the linear nature of the optimization problems under consideration, repeated interrogation of the objective and constraint functions is not necessary once gradient information is available. However, this reliance on linear approximations is not without difficulties. If the underlying problem is highly nonlinear, the algorithm may converge slowly. The presence of user-defined variable bounds may help the convergence if these bounds are properly chosen, such as through a trust region framework. Detailed knowledge of the design space can also help, but this increases the overhead cost of implementation.

Two other adaptations of the original BLISS architecture are known in the literature. The first is Ahn and Kwon's proBLISS [2], an architecture for reliability-based MDO. Their results show that the architecture is competitive with reliability-based adaptations of MDF and IDF. The second is LeGresley and Alonso's BLISS/POD [97], an architecture that integrates a reduced-order modeling technique called Proper Orthogonal Decomposition [13] to reduce the cost of the multidisciplinary analysis and sensitivity analysis steps. Their results show a significant improvement in the performance of BLISS, to the point where it is almost competitive with MDF.

As an enhancement of the original BLISS, a radically different formulation called BLISS-2000 was developed by Sobieski et al. [154]. Because BLISS-2000 does not require a multidisciplinary analysis to restore feasibility of the design, we have separated it from other BLISS variants in the classification shown in Fig. 7.13. In fact, like other IDF-derived architectures, BLISS-2000 uses coupling variable targets to enforce consistency at the optimum. Information exchange between system and discipline subproblems is completed through surrogate models of the disciplinary optima. The BLISS-2000 system subproblem is given by

$$\begin{aligned} \text{minimize} \quad & f_0\left(x, \tilde{y}\left(x, y^t\right)\right) \\ \text{with respect to} \quad & x_0, y^t, w \\ \text{subject to} \quad & c_0\left(x, \tilde{y}\left(x, y^t, w\right)\right) \geq 0 \\ & y_i^t - \tilde{y}_i\left(x_0, x_i, y_{j\neq i}^t, w_i\right) = 0 \quad \text{for} \quad i = 1, \ldots, N. \end{aligned} \tag{7.61}$$

The BLISS-2000 discipline i subproblem is

$$\begin{aligned} \text{minimize} \quad & w_i^T y_i \\ \text{with respect to} \quad & x_i \\ \text{subject to} \quad & c_i\left(x_0, x_i, y_i\left(x_0, x_i, y_{j\neq i}^t\right)\right) \geq 0. \end{aligned} \tag{7.62}$$

A unique aspect of this architecture is the use of a vector of weighting coefficients, w_i, attached to the disciplinary states. These weighting coefficients give the user a measure of control over state variable preferences. Generally speaking, the coefficients should be chosen based on the structure of the global objective to allow disciplinary subproblems to find an optimum more quickly. How much the choice of coefficients affects convergence has yet to be determined. The XDSM for the architecture is shown in Fig. 7.18 and the procedure is listed in Algorithm 20.

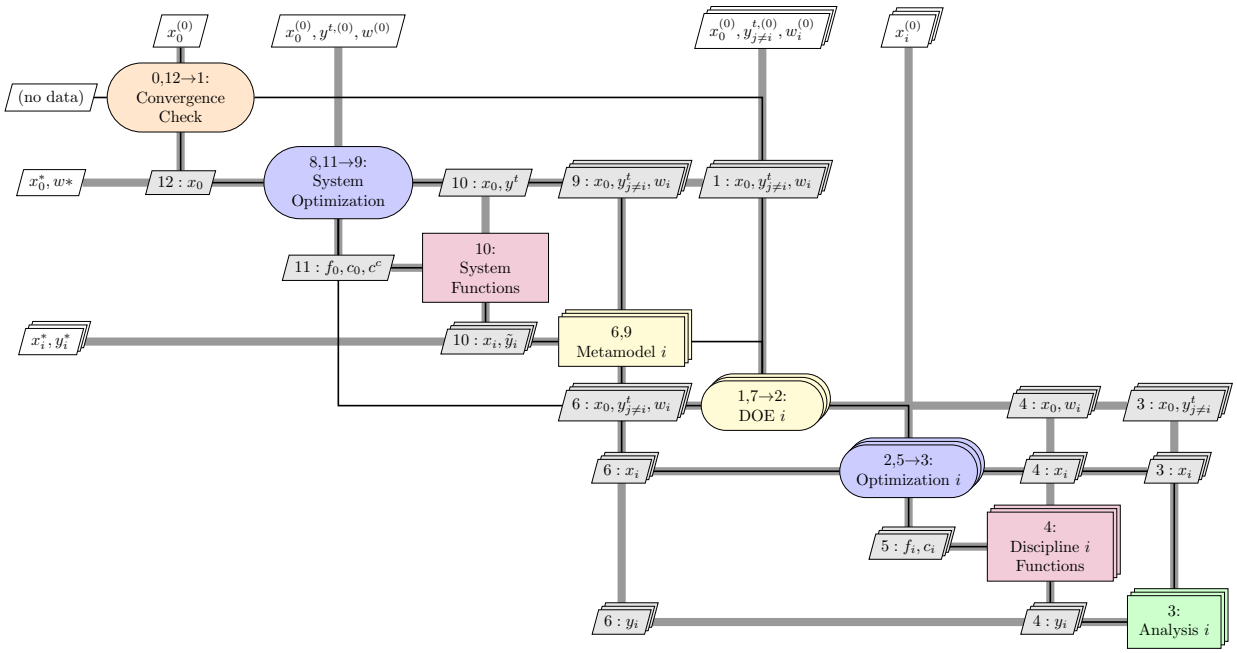


Figure 7.18: Diagram for the BLISS-2000 architecture

Algorithm 20 BLISS-2000

Input: Initial design variables x
Output: Optimal variables x^*, objective function f^*, and constraint values c^*

0: Initiate system optimization
repeat
  for Each discipline i **do**
    1: Initiate a DOE
    for Each DOE point **do**
      2: Initiate discipline subproblem optimization
      repeat
        3: Evaluate discipline analysis
        4: Compute discipline objective f_i and constraint functions c_i
        5: Update the local design variables x_i
      until $5 \rightarrow 3$: Disciplinary optimization subproblem has converged
      6: Update surrogate model of optimized disciplinary subproblem with new solution
    end for $7 \rightarrow 1$
  end for
  8: Initiate system subproblem optimization
  repeat
    9: Interrogate surrogate models with current values of system variables
    10: Compute system objective and constraint function values
    11: Compute a new system design point
  until $11 \rightarrow 9$ System subproblem has converged
until $12 \rightarrow 1$: System optimization has converged

BLISS-2000 possesses several advantages over the original BLISS architecture. First, the solution procedure is much easier to understand. Such simplicity greatly reduces the number of obstacles to implementation. Second, the decomposed problem formulation of BLISS-2000 is equivalent to Problem (7.18) provided that the weighting coefficients accurately capture the influence of the coupling variables on the objective [154]. Third, by using surrogate models for each discipline, rather than for the whole system, the calculations for BLISS-2000 can be run in parallel with minimal communication between disciplines. Finally, BLISS-2000 seems to be more flexible than its predecessor. Recently, Sobieski detailed an extension of BLISS-2000 to handle multilevel, system-of-systems problems [151]. In spite of these advantages, it appears that BLISS-2000 has not been used nearly as frequently as the original BLISS formulation.

Most of the benchmarking of BLISS used its original formulation. Many of the results available [39, 129, 171] suggest that BLISS is not competitive with other architectures in terms of computational cost. Concerns include both the number of discipline analyses required and the high cost of the post-optimality derivative computations. These problems can be mitigated by the use of surrogate models or reduced-order modeling as described above. The one benchmarking result available for BLISS-2000 is the launch vehicle design problem of Brown and Olds [26]. In this case, BLISS-2000 tended to outperform other distributed architectures and should be cost-competitive with other monolithic architectures when coarse-grained parallel processing is fully exploited. While this is a promising development, more work is needed to confirm the result.

7.6.6 Analytical Target Cascading (ATC)

The ATC architecture was not initially developed as an MDO architecture, but as a method to propagate system targets — i.e., requirements or desirable properties — through a hierarchical system to achieve a feasible system design satisfying these targets [81, 82]. If the system targets were unattainable, the ATC architecture would return a design point minimizing the unattainability. Effectively, then, the ATC architecture is no different from an MDO architecture with a system objective of minimizing the squared difference between a set of system targets and model responses. By simply changing the objective function, we can solve general MDO problems using ATC.

The ATC problem formulation that we present here is due to Tosserams et al. [162]. This formulation conforms to our definition of an MDO problem by explicitly including system wide objective and constraint functions. Like all other ATC problem formulations, a solution generated by this architecture will solve Problem (7.18) provided the penalization terms in the optimization problems all approach zero. The ATC system subproblem is given by

$$\begin{aligned} \text{minimize} \quad & f_0\left(x, y^t\right) + \sum_{i=1}^{N} \Phi_i\left(\hat{x}_{0i} - x_0, y_i^t - y_i\left(x_0, x_i, y^t\right)\right) + \\ & \Phi_0\left(c_0\left(x, y^t\right)\right) \\ \text{with respect to} \quad & x_0, y^t, \end{aligned} \tag{7.63}$$

where Φ_0 is a penalty relaxation of the global design constraints and Φ_i is a penalty relaxation of

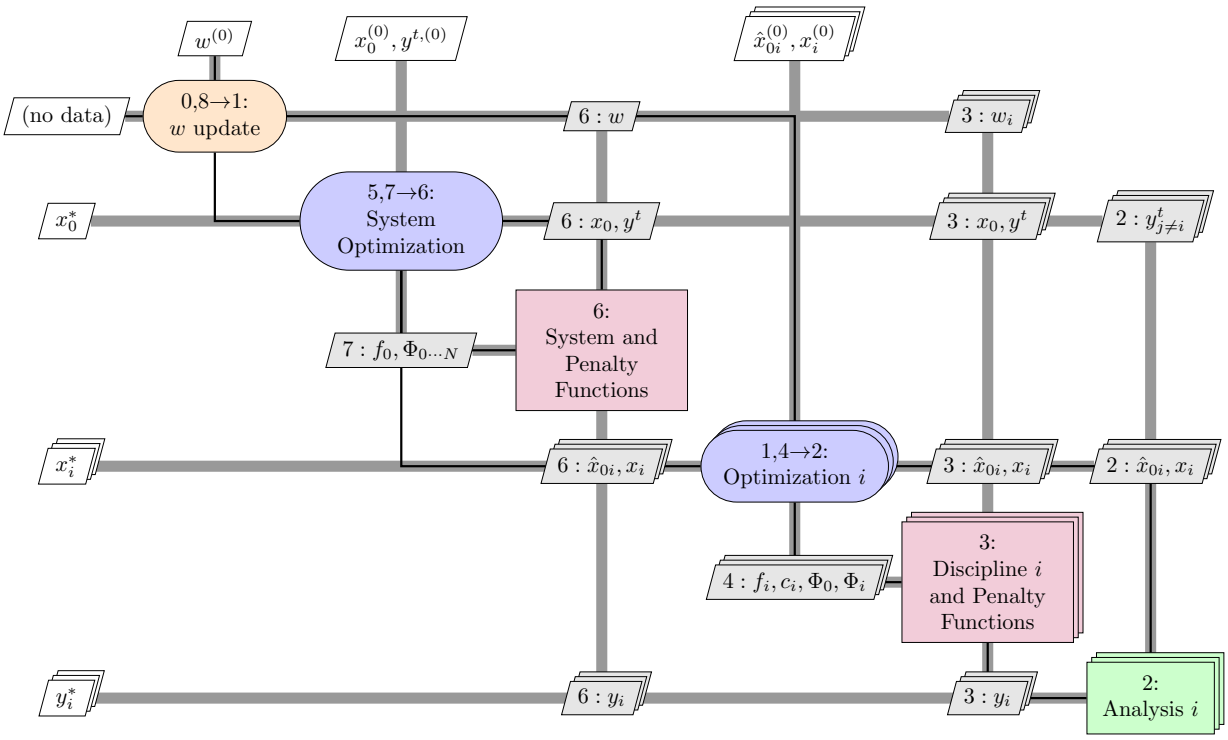


Figure 7.19: Diagram for the ATC architecture

the discipline i consistency constraints. The i^{th} discipline subproblem is:

$$\begin{aligned}
\text{minimize} \quad & f_0\left(\hat{x}_{0i}, x_i, y_i\left(\hat{x}_{0i}, x_i, y^t_{j\neq i}\right), y^t_{j\neq i}\right) + f_i\left(\hat{x}_{0i}, x_i, y_i\left(\hat{x}_{0i}, x_i, y^t_{j\neq i}\right)\right) + \\
& \Phi_i\left(y_i^t - y_i\left(\hat{x}_{0i}, x_i, y^t_{j\neq i}\right), \hat{x}_{0i} - x_0\right) + \\
& \Phi_0\left(c_0\left(\hat{x}_{0i}, x_i, y_i\left(\hat{x}_{0i}, x_i, y^t_{j\neq i}\right), y^t_{j\neq i}\right)\right) \\
\text{with respect to} \quad & \hat{x}_{0i}, x_i \\
\text{subject to} \quad & c_i\left(\hat{x}_{0i}, x_i, y_i\left(\hat{x}_{0i}, x_i, y^t_{j\neq i}\right)\right) \geq 0.
\end{aligned} \tag{7.64}$$

Fig. 7.19 shows the ATC architecture XDSM, where w denotes the penalty function weights used in the determination of Φ_0 and Φ_i. The details of ATC are described in Algorithm 21.

Algorithm 21 ATC

Input: Initial design variables x
Output: Optimal variables x^*, objective function f^*, and constraint values c^*

```
0: Initiate main ATC iteration
repeat
  for Each discipline i do
    1: Initiate discipline optimizer
    repeat
      2: Evaluate disciplinary analysis
      3: Compute discipline objective and constraint functions and penalty function values
      4: Update discipline design variables
    until 4 → 2: Discipline optimization has converged
  end for
  5: Initiate system optimizer
  repeat
    6: Compute system objective, constraints, and all penalty functions
    7: Update system design variables and coupling targets.
  until 7 → 6: System optimization has converged
  8: Update penalty weights
until 8 → 1: Penalty weights are large enough
```

ATC can be applied to a multilevel hierarchy of systems just as well as a discipline-based nonhierarchic system. In the multilevel case, the penalty functions are applied to all constraints that combine local information with information from the levels immediately above or below the current one. Post-optimality derivatives are not needed in any of the subproblems, as nonlocal data are always treated as fixed values in the current subproblem. Furthermore, as with penalty methods for general optimization problems, (see, e.g., Nocedal and Wright [123, chap. 17]) the solution of the MDO problem must be computed to reasonable accuracy before the penalty weights are updated. However, because we are now dealing with a distributed set of subproblems, the whole hierarchy of subproblems must be solved for a given set of weights. This is due to the nonseparable nature of the quadratic penalty function.

The most common penalty functions in ATC are quadratic penalty functions. In this case, the proper selection of the penalty weights is important for both final consistency in the discipline models and convergence of the algorithm. Michalek and Papalambros [117] present an effective weight update method that is especially useful when unattainable targets have been set in a traditional ATC process. Michelena et al. [122] present several coordination algorithms using ATC with quadratic penalty functions and demonstrate the convergence for all of them. However, the analysis of Michelena et al. is based on a top-level target satisfaction objective function that is convex. The behavior of those versions of ATC in the presence of nonconvex objectives is unknown.

Several other penalty function choices and associated coordination approaches have also been devised for ATC. Kim et al. [79] outline a version of ATC that uses Lagrangian relaxation and a sub-gradient method to update the multiplier estimates. Tosserams et al. [160] use augmented Lagrangian relaxation with Bertsekas' method of multipliers [14] and alternating direction method of multipliers [15] to update the penalty weights. They also group this variant of ATC into a larger class of coordination algorithms known as Augmented Lagrangian Coordination [162]. Li et al. [99] apply the diagonal quadratic approximation approach of Ruszcynski [138] to the augmented Lagrangian to eliminate subproblem coupling through the quadratic terms and further parallelize

the architecture. Finally, Han and Papalambros [63] propose a version of ATC based on sequential linear programming [11, 30], where inconsistency is penalized using infinity norms. They later presented a convergence proof of this approach in a short note [64]. For each of the above penalty function choices, ATC was able to produce the same design solutions as the monolithic architectures.

Despite having been developed relatively recently, the ATC architecture has been widely used. By far, ATC has been most frequently applied to design problems in the field for which it was developed, the automotive industry [21, 31, 65, 78, 83, 89, 90, 156]. However, the ATC approach has also proven to be useful in aircraft design [6, 7, 167] and building design [34]. ATC has also found applications outside of strict engineering design problems, including manufacturing decisions [100], supply chain management [69], and marketing decisions in product design [119]. Huang et al. [70] have developed an ATC-specific web portal to solve optimization problems via the ATC architecture. Etman et al. [45] discuss the automatic implementation of coordination procedures, using ATC as an example architecture. ATC formulations exist that can handle integer variables [118] and probabilistic design problems [91, 103]. Another important adaptation of ATC applies to problems with block-separable linking constraints [164]. In this class of problems, c_0 consists of constraints that are sums of functions depending on only shared variables and the local variables of one discipline.

The performance of ATC compared with other architectures is not well known, because only one result is available. In de Wit and Van Keulen's architecture comparison [39], ATC is competitive with all other benchmarked distributed architectures, including standard versions of CSSO, CO, and BLISS. However, using a gradient-based optimizer, ATC and the other distributed architectures are not competitive with a monolithic architecture in terms of the number of function and discipline evaluations. More commonly, different versions of ATC are benchmarked against each other. Tosserams et al. [160] compared the augmented Lagrangian penalty approach with the quadratic penalty approach and found much improved results with the alternating direction method of multipliers. Surprisingly, de Wit and van Keulen [39] found the augmented Lagrangian version performed worse than the quadratic penalty method for their test problem. Han and Papalambros [63] compared their sequential linear programming version of ATC to several other approaches and found a significant reduction in the number of function evaluations. However, they note that the coordination overhead is large compared to other ATC versions and still needs to be addressed.

7.6.7 Exact and Inexact Penalty Decomposition (EPD and IPD)

If there are no system-wide constraints or objectives, i.e., if neither f_0 and c_0 exist, the Exact or Inexact Penalty Decompositions (EPD or IPD) [41, 43] may be employed. Both formulations rely on solving the disciplinary subproblem

$$\begin{aligned} \text{minimize} \quad & f_i\left(\hat{x}_{0i}, x_i, y_i\left(\hat{x}_{0i}, x_i, y^t_{j\neq i}\right)\right) + \Phi_i\left(\hat{x}_{0i} - x_0, y^t_i - y_i\left(\hat{x}_{0i}, x_i, y^t_{j\neq i}\right)\right) \\ \text{with respect to} \quad & \hat{x}_{0i}, x_i \\ \text{subject to} \quad & c_i\left(\hat{x}_{0i}, x_i, y_i\left(x_{0i}, x_i, y^t_{j\neq i}\right)\right) \geq 0. \end{aligned} \tag{7.65}$$

Here, Φ_i denotes the penalty function associated with the inconsistency between the i^{th} discipline's information and the system information. In EPD, Φ_i is an L_1 penalty function with additional variables and constraints added to ensure smoothness. In IPD, Φ_i is a quadratic penalty function with appropriate penalty weights. The notation $\hat{x}_{0i}$ denotes a local copy of the shared design variables in discipline i, while x_0 denotes the system copy.

At the system level, the subproblem is an unconstrained minimization with respect to the target variables. The objective function is the sum of the optimized disciplinary penalty terms, denoted

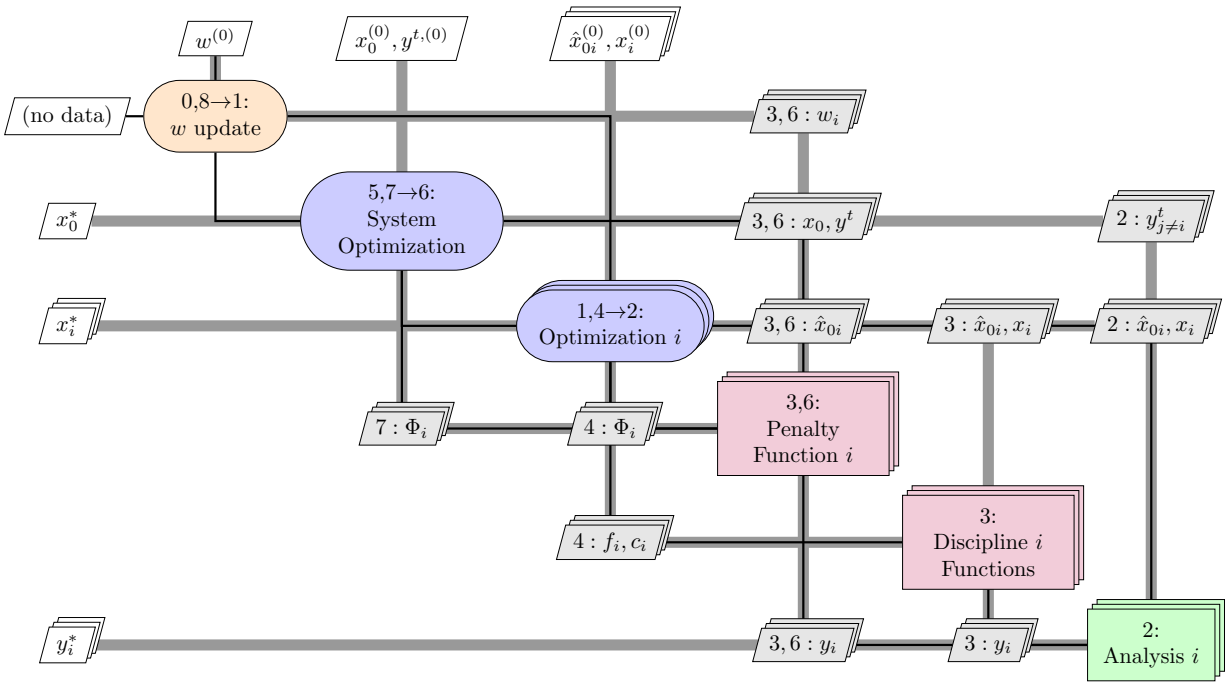


Figure 7.20: Diagram for the penalty decomposition architectures EPD and IPD

as Φ_i^*.

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{N} \Phi_i^* \left(x_0, y^t\right) = \sum_{i=1}^{N} \Phi_i \left(\hat{x}_{0i} - x_0, y_i^t - y_i \left(\hat{x}_{0i}, x_i, y^t_{j\neq i}\right)\right) \\ \text{with respect to} \quad & x_0, y^t \end{aligned} \tag{7.66}$$

The penalty weights are updated upon solution of the system problem. Fig. 7.20 shows the XDSM for this architecture, where w represents the penalty weights. The sequence of operations in this architecture is detailed in Algorithm 22.

Algorithm 22 EPD and IPD

Input: Initial design variables x
Output: Optimal variables x^*, objective function f^*, and constraint values c^*
0: Initiate main iteration
repeat
  for Each discipline i **do**
    repeat
      1: Initiate discipline optimizer
      2: Evaluate discipline analysis
      3: Compute discipline objective and constraint functions, and penalty function values
      4: Update discipline design variables
    until $4 \rightarrow 2$: Discipline optimization has converged
  end for
  5: Initiate system optimizer
  repeat
    6: Compute all penalty functions
    7: Update system design variables and coupling targets
  until $7 \rightarrow 6$: System optimization has converged
  8: Update penalty weights.
until $8 \rightarrow 1$: Penalty weights are large enough

Both EPD and IPD have mathematically provable convergence to a KKT point under the linear independence constraint qualification and with mild assumptions on the update strategy for the penalty weights [43]. In particular, the penalty weight in IPD must monotonically increase until the inconsistency is sufficiently small, similar to other quadratic penalty methods [123]. For EPD, the penalty weight must be larger than the largest Lagrange multiplier, following established theory of the L_1 penalty function [123], while the barrier parameter must monotonically decrease like in an interior point method [169]. If other penalty functions are employed, the parameter values are selected and updated according to the corresponding mathematical theory. Furthermore, under these conditions, the solution obtained under EPD and IPD will also be a solution to Problem (7.18).

Only once in the literature has either penalty decomposition architecture been tested against any others. The results of Tosserams et al. [161] suggest that performance depends on the choice of penalty function employed. A comparison between IPD with a quadratic penalty function and IPD with an augmented Lagrangian penalty function showed that the latter significantly outperformed the former in terms of both time and number of function evaluations on several test problems.

7.6.8 MDO of Independent Subspaces (MDOIS)

If the problem contains no system-wide constraints or objectives, i.e., if neither f_0 and c_0 exist, and the problem does not include shared design variables, i.e., if x_0 does not exist, then the MDO of independent subspaces (MDOIS) architecture [146] applies. In this case, the discipline subproblems are fully separable (aside from the coupled state variables) and given by

$$\begin{aligned} \text{minimize} \quad & f_i\left(x_i, y_i\left(x_i, y^t_{j\neq i}\right)\right) \\ \text{with respect to} \quad & x_i \\ \text{subject to} \quad & c_i\left(x_i, y_i\left(x_i, y^t_{j\neq i}\right)\right) \geq 0. \end{aligned} \tag{7.67}$$

In this case, the coupling targets are just parameters representing system state information. Upon

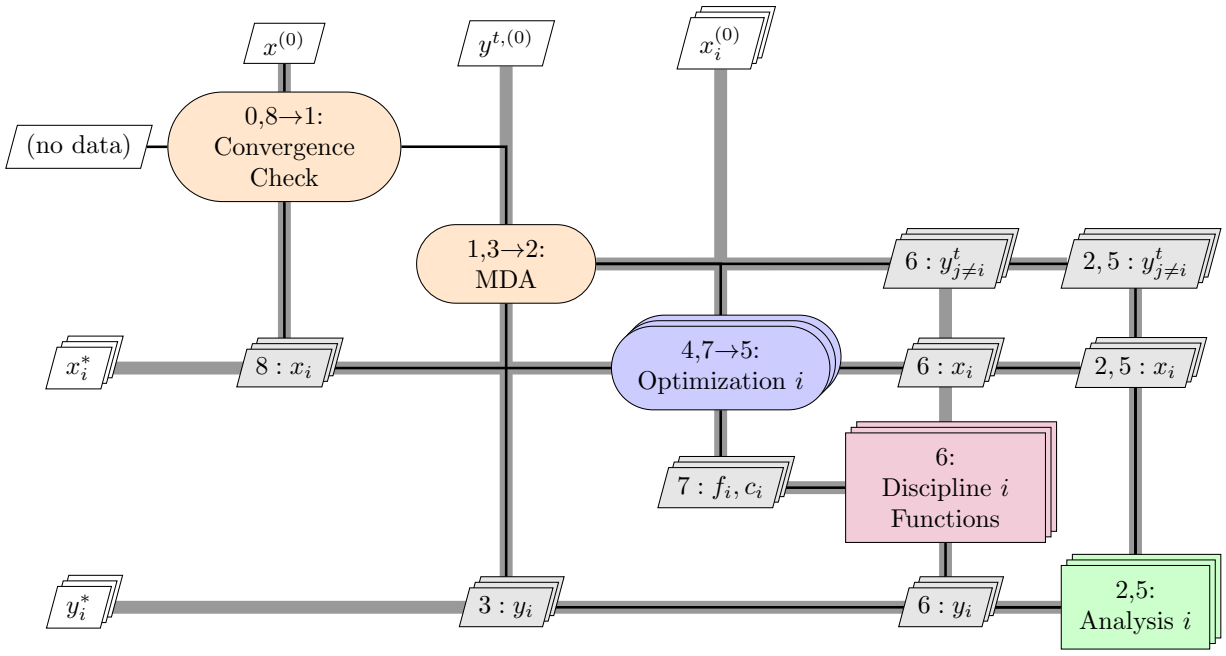


Figure 7.21: Diagram for the MDOIS architecture

solution of the disciplinary problems, which can access the output of individual disciplinary analysis codes, a full multidisciplinary analysis is completed to update all target values. Thus, rather than a system subproblem used by other architectures, the MDA is used to guide the disciplinary subproblems to a design solution. Shin and Park [146] show that under the given problem assumptions an optimal design is found using this architecture. Fig. 7.21 depicts this process in an XDSM. MDOIS is detailed in Algorithm 23.

Algorithm 23 MDOIS

Input: Initial design variables x
Output: Optimal variables x^*, objective function f^*, and constraint values c^*

```
0: Initiate main iteration
repeat
  repeat
    1: Initiate MDA
    2: Evaluate discipline analyses
    3: Update coupling variables
  until 3 → 2: MDA has converged
  for Each discipline i do
    4: Initiate disciplinary optimization
    repeat
      5: Evaluate discipline analysis
      6: Compute discipline objectives and constraints
      7: Compute a new discipline design point
    until 7 → 5: Discipline optimization has converged
  end for
until 8 → 1 Main iteration has converged
```

Benchmarking results are available comparing MDOIS to some of the older architectures. These results are given by Yi et al. [171]. In many cases, MDOIS requires fewer analysis calls than MDF while still being able to reach the optimal solution. However, MDOIS still does not converge as fast as IDF and the restrictive definition of the problem means that the architecture is not nearly as flexible as MDF. A practical problem that can be solved using MDOIS is the belt-integrated seat problem of Shin et al. [145]. However, the results using MDOIS have not been compared to results obtained using other architectures.

7.6.9 Quasiseparable Decomposition (QSD)

Haftka and Watson [61] developed the QSD architecture to solve *quasiseparable* optimization problems. In a quasiseparable problem, the system objective and constraint functions are assumed to be dependent only on global variables (i.e., the shared design and coupling variables). This type of problem may be thought of as identical to the complicating variables problems discussed in Section 7.6.1. In our experience, we have not come across any practical design problems that do not satisfy this property. However, if required by the problem, we can easily transform the general MDO problem (7.18) into a quasiseparable problem. This is accomplished by duplicating the relevant local variables, and forcing the global objective to depend on the duplicates. (The process is analogous to adapting the general MDO problem to the original Collaborative Optimization architecture.) Provided the duplicated variables converge to the same value, a solution to the quasiseparable problem is also a solution to the original problem. The system subproblem of QSD

is given by

$$\begin{aligned} \text{minimize} \quad & f_0\left(x_0, y^t\right) + \sum_{i=1}^{N} b_i \\ \text{with respect to} \quad & x_0, y^t, b \\ \text{subject to} \quad & c_0\left(x_0, y^t\right) \geq 0 \\ & s_i^*\left(x_0, x_i, y_i\left(x_0, x_i, y_{j\neq i}^t\right), b_i\right) \geq 0 \quad \text{for} \quad i = 1, \ldots, N. \end{aligned} \tag{7.68}$$

where s_i is the constraint margin for discipline i and b_i is the "budget" assigned to each disciplinary objective. The discipline i subproblem becomes

$$\begin{aligned} \text{minimize} \quad & -s_i \\ \text{with respect to} \quad & x_i, s_i \\ \text{subject to} \quad & c_i\left(x_0, x_i, y_i\left(x_0, x_i, y_{j\neq i}^t\right)\right) - s_i \geq 0 \\ & f_i\left(x_0, x_i, y_i\left(x_0, x_i, y_{j\neq i}^t\right)\right) - b_i - s_i \geq 0 \\ & y_i^t - y_i\left(x_0, x_i, y_{j\neq i}^t\right) = 0 \end{aligned} \tag{7.69}$$

where k is an element of the constraint vector c_i. Due to the use of target coupling variables, we classify this architecture as distributed IDF. Similarly to CO, this is a bilevel architecture where the solutions of disciplinary subproblems are constraints in the system subproblem. Therefore, post-optimality derivatives or surrogate model approximations of optimized disciplinary subproblems are required to solve the system subproblem. The standard architecture is shown as an XDSM in Fig. 7.22. The sequence of operations in QSD is given by Algorithm 24. Haftka and Watson have extended the theory behind QSD to solve problems with a combination of discrete and continuous variables [62]. Liu et al. [102] successfully applied QSD with surrogate models to a structural optimization problem. However, they made no comparison of the performance to other architectures, not even QSD without the surrogates. A version of QSD without surrogate models was benchmarked by de Wit and van Keulen [39]. Unfortunately, this architecture was the worst of all the architectures tested in terms of disciplinary evaluations. A version of QSD using surrogate models should yield improved performance, due to the smoothness introduced by the model, but this version has not been benchmarked to our knowledge.

Algorithm 24 QSD

Input: Initial design variables x
Output: Optimal variables x^*, objective function f^*, and constraint values c^*

```
0: Initiate system optimization
repeat
  1: Compute system objectives and constraints
  for Each discipline i do
    1.0: Initiate discipline optimization
    repeat
      1.1: Evaluate discipline analysis
      1.2: Compute discipline objective and constraints
      1.3: Update discipline design point
    until 1.3 → 1.1: Discipline optimization has converged
  end for
  2: Compute a new system design point
until 2 → 1: System problem has converged
```

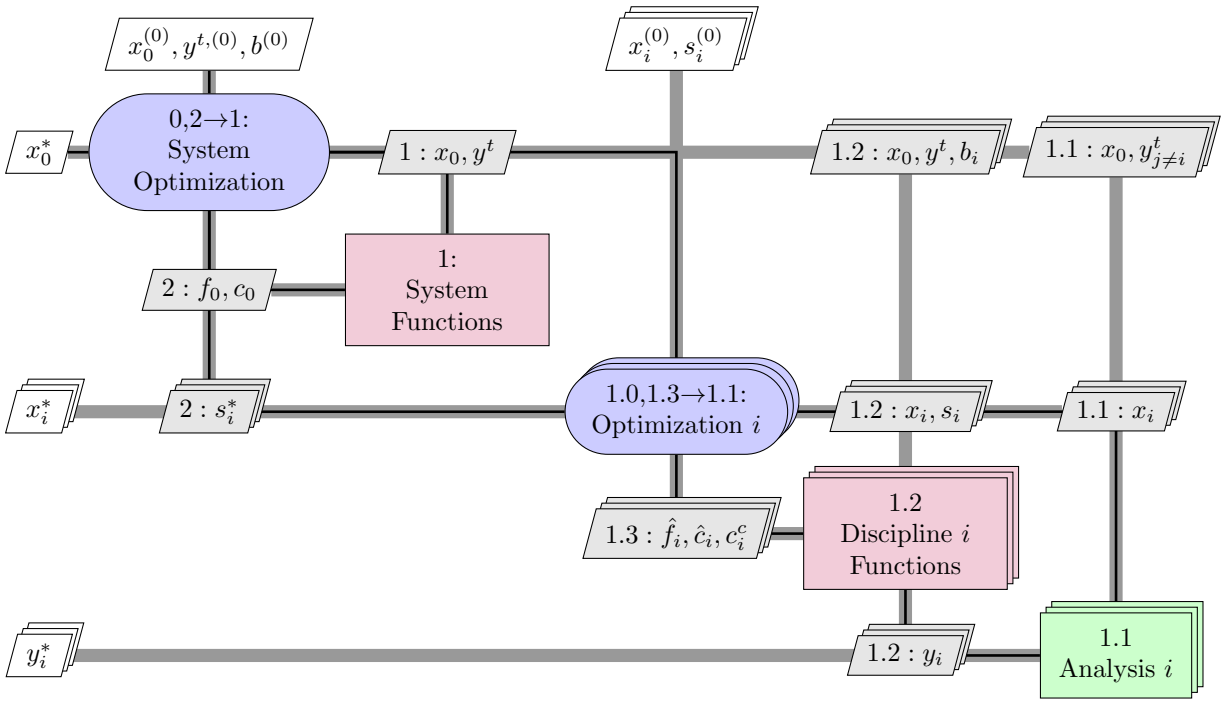


Figure 7.22: Diagram for the QSD architecture

7.6.10 Asymmetric Subspace Optimization (ASO)

The ASO architecture [33] is a new distributed-MDF architecture. It was motivated by the case of high-fidelity aerostructural optimization, where the aerodynamic analysis typically requires an order of magnitude more time to complete than the structural analysis [111]. To reduce the number of expensive aerodynamic analyses, the structural analysis is coupled with a structural optimization inside the MDA. This idea can be readily generalized to any problem where there is a wide discrepancy between discipline analysis times. Fig. 7.23 shows the optimization of the third discipline of a generic problem within the ASO architecture. The sequence of operations in ASO is listed in Algorithm 25.

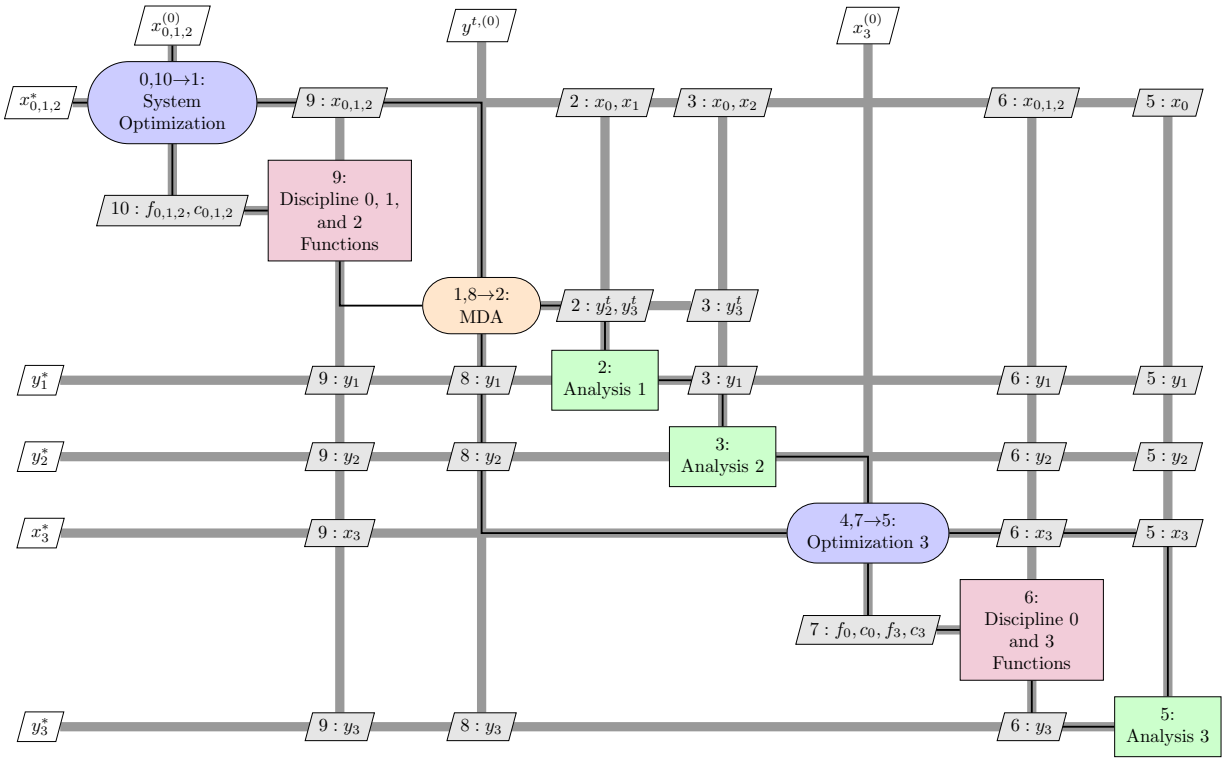


Figure 7.23: Diagram for the ASO architecture

Algorithm 25 ASO

Input: Initial design variables x
Output: Optimal variables x^*, objective function f^*, and constraint values c^*
0: Initiate system optimization
repeat
1: Initiate MDA
repeat
2: Evaluate Analysis 1
3: Evaluate Analysis 2
4: Initiate optimization of Discipline 3
repeat
5: Evaluate Analysis 3
6: Compute discipline 3 objectives and constraints
7: Update local design variables
until 7 $\rightarrow$ 5: Discipline 3 optimization has converged
8: Update coupling variables
until 8 $\rightarrow$ 2 MDA has converged
9: Compute objective and constraint function values for all disciplines 1 and 2
10: Update design variables
until 10 $\rightarrow$ 1: System optimization has converged

The optimality of the final solution is preserved by using the coupled post-optimality sensitivity (CPOS) equations, developed by Chittick and Martins [33], to calculate gradients at the system level. CPOS represents the extension of the coupled sensitivity equations [112, 152] to include the optimality conditions. ASO was later implemented using a coupled-adjoint approach as well [32]. Kennedy et al. [76] present alternative strategies for computing the disciplinary subproblem optima and the post-optimality sensitivity analysis. As with other bilevel MDO architectures, the post-optimality analysis is necessary to ensure convergence to an optimal design of the original monolithic problem. We also note the work of Kaufman et al. [74] and Hosder et al. [67], who use structural optimization as an inner loop in aircraft design optimization with surrogate models.

The system subproblem in ASO is

$$\begin{aligned} \text{minimize} \quad & f_0\left(x, y\left(x, y\right)\right) + \sum_k f_k\left(x_0, x_k, y_k\left(x_0, x_k, y_{j\neq k}\right)\right) \\ \text{with respect to} \quad & x_0, x_k \\ \text{subject to} \quad & c_0\left(x, y\left(x, y\right)\right) \geq 0 \\ & c_k\left(x_0, x_k, y_k\left(x_0, x_k, y_{j\neq k}\right)\right) \geq 0 \qquad \text{for all } k, \end{aligned} \tag{7.70}$$

where subscript k denotes disciplinary information that remains outside of the MDA. The disciplinary problem for discipline i, which is resolved inside the MDA, is

$$\begin{aligned} \text{minimize} \quad & f_0\left(x, y\left(x, y\right)\right) + f_i\left(x_0, x_i, y_i\left(x_0, x_i, y_{j\neq i}\right)\right) \\ \text{with respect to} \quad & x_i \\ \text{subject to} \quad & c_i\left(x_0, x_i, y_i\left(x_0, x_i, y_{j\neq i}\right)\right) \geq 0. \end{aligned} \tag{7.71}$$

The results show a substantial reduction in the number of calls to the aerodynamics analysis, and even a slight reduction in the number of calls to the structural analysis [33]. However, the total time for the optimization routine is only competitive with MDF if the aerodynamic analysis

is substantially more costly than the structural analysis. If the two analyses take roughly equal time, MDF is still much faster. Furthermore, the use of CPOS increases the complexity of the sensitivity analysis compared to a normal coupled adjoint, which adds to the overall solution time. As a result, this architecture may only appeal to practitioners solving MDO problems with widely varying computational cost in the discipline analysis.

7.7 Benchmarking of MDO Architectures

One of the challenges facing the MDO community is determining which architecture is most efficient for a given MDO problem or class of problems. One way this can be accomplished is by benchmarking the architectures using simple test problems. Many authors developing new architectures include a few results in their work. However, it is especially important to test multiple architectures on a given problem, as the most appropriate architecture often depends on the nature of the problem itself. A number of studies specifically focused on benchmarking are available in the literature [8, 26, 72, 84, 87, 129, 159, 171]. However, there are a number of limitations in these studies that need to be considered when reading them and considering future work.

The first limitation is the fact that no two studies are likely to produce identical results. This stems from the fact that no two programmers can be expected to program the same architecture the same way. In addition, experts in one particular architecture may be able to more fully optimize the performance of their architecture, unintentionally biasing the results [171]. This problem can be overcome by performing the benchmarking in a specialized MDO framework such as iSight, ModelCenter, pyMDO [113], or OpenMDAO [52]. These frameworks allow single definitions of the MDO problem and the architecture to be frequently reused, eliminating most of the "human factor" from the optimization. Padula and Gillian [124] present a brief history of frameworks for MDO. Kodiyalam and Sobieski [86] and Kodiyalam et al. [88] discuss requirements for an MDO framework and high-performance computing considerations respectively.

The second limitation is the choice of architectures used in the benchmarking studies. Most studies tend to focus on the most mature architectures, especially MDF, IDF, CO, and CSSO. This should become less of a limitation with time as the newer architectures become better known. We must emphasize, though, that implementing the new architecture in an established MDO framework should allow more rapid benchmarking of the new architecture and give a much earlier indication of the architecture's promise.

The third limitation is in the test problems themselves. Almost all of the test problems are of low dimensionality, and many have discipline analyses consisting of analytic functions. In high-fidelity design, the MDO problem could have thousands of variables and constraints with discipline analyses that take minutes or hours to evaluate, even using high-performance parallel computers. While the test problems used in benchmarking may never reach this level of complexity, they should be large enough to establish a clear trend in terms of time and number of function calls. Ideally, these problems would also be scalable to examine the effects of problem dimensionality on architecture efficiency. An early attempt at such a scalable problem was presented by Tedford and Martins [159] and allowed the user to define an arbitrary number of disciplines, design variables, and coupling variables. Another benchmarking problem that possesses some scalability is the microaccelerometer problem of Tosserams et al. [166] Test problems with lower dimensionality but with multiple local minima are presented by Tosserams et al. [165] We also feel that creating a test suite of MDO problems similar to the now-defunct NASA MDO suite [125] would be of great utility to researchers and practitioners in the field.

7.8 Analytic Methods for Computing Coupled Derivatives

We now extend the analytic methods derived in Chapter 4 to multidisciplinary systems, where each discipline is seen as one component. We start with the equations in the last row of Fig. 4.23 (also repeated in the first row of Fig. 7.26), which represent the direct and adjoint methods, respectively, for a given system. In this form, the Jacobian matrices for the direct and adjoint methods are block lower and upper triangular, respectively, but not fully lower or upper triangular because we eliminate the inverse of the Jacobian $\partial \boldsymbol{R}/\partial \boldsymbol{y}$ at the expense of creating a linear system that must now be solved. This inverse Jacobian is necessary to obtain an explicit definition for $\boldsymbol{y}$, but we eliminate it because the solution of a linear system is cheaper than matrix inversion of the same size.

The direct and adjoint methods for multidisciplinary systems can be derived by partitioning the various variables by disciplines, as follows,

$$\boldsymbol{R} = [\boldsymbol{R}_1, \boldsymbol{R}_2]^T \quad \boldsymbol{y} = [\boldsymbol{y}_1, \boldsymbol{y}_2]^T \tag{7.72}$$

where we have assumed two different disciplines. All of the design variables are included in $\boldsymbol{x}$. Then, we use these vectors in the equations in the direct and adjoint equations shown in the first row of Fig. 7.26 to obtain the second row in Fig. 7.26. These are the coupled versions of the direct and adjoint methods, respectively. The coupled direct method was first developed by Bloebaum and Sobieski [17, 20, 152], while the coupled adjoint was originally developed by Martins et al. [112]. The both the coupled direct and adjoint methods have since been applied to the aerostructural design optimization of aircraft wings [77, 111, 114**?**].

Fig. 7.24 illustrates the level of decomposition that this involves, using earlier notation. In Fig. 7.26, we can see that this decomposition turns the Jacobian $\partial \boldsymbol{R}/\partial \boldsymbol{y}$ into a matrix with distinct blocks corresponding to the different disciplines. Fig. 7.25(a) shows a graphical view of the two-discipline system.

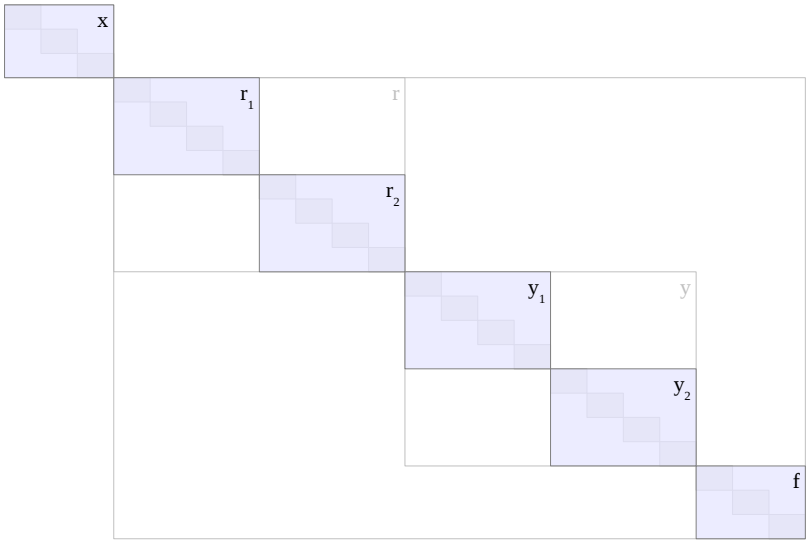
Fig. 7.25(b) and the third row in Fig. 7.26 show another alternative for obtaining the total derivatives of multidisciplinary systems that was first developed by Sobieski [152] for the direct method, and by Martins et al. [112] for the adjoint method. The advantage of this approach is that we do not need to know the residuals of a given disciplinary solvers, but instead can use the *coupling variables*. To derive the direct and adjoint versions of this approach within our mathematical framework, we define the artificial residual functions

$$\boldsymbol{R}_i = \boldsymbol{Y}_i - \boldsymbol{y}_i, \tag{7.73}$$

where the $\boldsymbol{y}_i$ vector contains the intermediate variables of the i^{th} discipline, and $\boldsymbol{Y}_i$ is the vector of functions that explicitly define these intermediate variables. This leads to the third row of equations in Fig. 7.26, which we call the *functional* approach. This contrasts with the *residual* approach that we used previously.

The $\boldsymbol{y}_i$ vector is treated as a vector of state variables in the terminology of systems with residuals. In general, the functions in the vector $\boldsymbol{Y}_i$ can depend on all other intermediate variables in the i^{th} discipline as well as any other disciplines, signifying that this development allows for coupling to be present among the intermediate variables.

To further generalize the computation of derivatives for multidisciplinary systems, consider a system with two disciplines: one with residuals and one without, as shown in Fig. 7.25(c). In a practical setting, this could correspond to a problem with one discipline that has residuals that are nonlinear in its state variables and another discipline where all intermediate variables are explicitly defined. Because of the high cost of the first discipline, it would be valuable to be able to use



$$\boldsymbol{v} = [\underbrace{v_1, \ldots, v_{n_x}}_{\boldsymbol{x}}, \underbrace{\ldots, v_{(n_x+n_{y1})}}_{\boldsymbol{r_1}}, \underbrace{\ldots, v_{(n_x+n_{y1}+n_{y2})}}_{\boldsymbol{r_2}}, \underbrace{\ldots, v_{(n_x+2n_{y1}+n_{y2})}}_{\boldsymbol{y_1}}, \underbrace{\ldots, v_{(n_x+2n_{y1}+2n_{y2})}}_{\boldsymbol{y_2}}, \underbrace{v_{(n-n_f)}, \ldots, t_n}_{\boldsymbol{f}}]^T.$$

Figure 7.24: Decomposition for the disciplinary level

the direct or adjoint method even with the second discipline added. Eqs. (g) and (h) in Fig. 7.26 show that this is possible with a hybrid formulation that combines elements from the residual and functional approaches. Eqs. (g) and (h) can be generalized to any number of disciplines with a combination of residuals (using the residual approach) and explicit intermediate variables (using the functional approach).

Example 7.43. Numerical Example

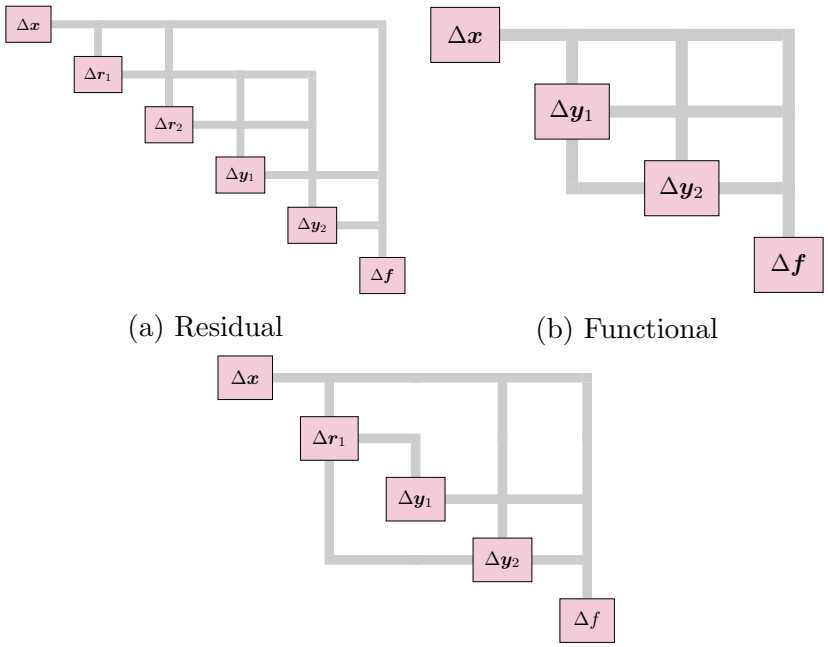
For the purpose of illustrating the coupled analytic methods, we now interpret the numerical example from Section 13 as a multidisciplinary system. In this view, the state variable y_1 belongs to discipline 1 and is solely constrained by residual R_1, while y_2 and R_2 are corresponding variables associated with discipline 2.

It should be obvious from Fig. 7.26 that the residual approach yields the same equations as the analytic methods except for the classification of blocks corresponding to different disciplines in the former case. In this example, each block is 1×1 since both disciplines have only one state variable.

For the functional approach, the residuals are eliminated from the equations and it is assumed that explicit expressions can be found for all state variables in terms of input variables and state variables from other disciplines. In most cases, the explicit computation of state variables involves solving the nonlinear system corresponding to the discipline; however, in this example, this is simplified because the residuals are linear in the state variables and each discipline has only one state variable. Thus, the explicit forms are

$$Y_1(x_1, x_2, y_2) = -\frac{2y_2}{x_1} + \frac{\sin x_1}{x_1} \tag{7.74}$$

$$Y_2(x_1, x_2, y_1) = \frac{y_1}{x_2^2}. \tag{7.75}$$



(a) Residual (b) Functional

(c) Hybrid

Figure 7.25: The different approaches for handling coupled multidisciplinary systems

Fig. 7.27 shows how df_1/dx_1 can be computed using the coupled residual, functional, and hybrid approaches, which correspond to cases in which neither, both, or just discipline 2 has this explicit form.

$$\begin{bmatrix} \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} \\ -\dfrac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}} & -\dfrac{\partial \boldsymbol{R}}{\partial \boldsymbol{y}} & \boldsymbol{0} \\ -\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}} & -\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{I} \\ \dfrac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}\boldsymbol{x}} \\ \dfrac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{I} \\ \boldsymbol{0} \\ \boldsymbol{0} \end{bmatrix}$$

(a) Direct method

$$\begin{bmatrix} \boldsymbol{I} & -\left[\dfrac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}}\right]^T & -\left[\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right]^T \\ \boldsymbol{0} & -\left[\dfrac{\partial \boldsymbol{R}}{\partial \boldsymbol{y}}\right]^T & -\left[\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}}\right]^T \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \dfrac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} \\ \dfrac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{r}} \\ \boldsymbol{I} \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{I} \end{bmatrix}$$

(b) Adjoint method

$$\begin{bmatrix} \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ -\dfrac{\partial \boldsymbol{R}_1}{\partial \boldsymbol{x}} & -\dfrac{\partial \boldsymbol{R}_1}{\partial \boldsymbol{y}_1} & -\dfrac{\partial \boldsymbol{R}_1}{\partial \boldsymbol{y}_2} & \boldsymbol{0} \\ -\dfrac{\partial \boldsymbol{R}_2}{\partial \boldsymbol{x}} & -\dfrac{\partial \boldsymbol{R}_2}{\partial \boldsymbol{y}_1} & -\dfrac{\partial \boldsymbol{R}_2}{\partial \boldsymbol{y}_2} & \boldsymbol{0} \\ -\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}} & -\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}_1} & -\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}_2} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{I} \\ \dfrac{\mathrm{d}\boldsymbol{y}_1}{\mathrm{d}\boldsymbol{x}} \\ \dfrac{\mathrm{d}\boldsymbol{y}_2}{\mathrm{d}\boldsymbol{x}} \\ \dfrac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{I} \\ \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{0} \end{bmatrix}$$

(c) Coupled direct — residual approach

$$\begin{bmatrix} \boldsymbol{I} & -\left[\dfrac{\partial \boldsymbol{R}_1}{\partial \boldsymbol{x}}\right]^T & -\left[\dfrac{\partial \boldsymbol{R}_2}{\partial \boldsymbol{x}}\right]^T & -\left[\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right]^T \\ \boldsymbol{0} & -\left[\dfrac{\partial \boldsymbol{R}_1}{\partial \boldsymbol{y}_1}\right]^T & -\left[\dfrac{\partial \boldsymbol{R}_2}{\partial \boldsymbol{y}_1}\right]^T & -\left[\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}_1}\right]^T \\ \boldsymbol{0} & -\left[\dfrac{\partial \boldsymbol{R}_1}{\partial \boldsymbol{y}_2}\right]^T & -\left[\dfrac{\partial \boldsymbol{R}_2}{\partial \boldsymbol{y}_2}\right]^T & -\left[\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}_2}\right]^T \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \dfrac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} \\ \dfrac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{r}_1} \\ \dfrac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{r}_2} \\ \boldsymbol{I} \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{I} \end{bmatrix}$$

(d) Coupled adjoint — residual approach

$$\begin{bmatrix} \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ -\dfrac{\partial \boldsymbol{Y}_1}{\partial \boldsymbol{x}} & \boldsymbol{I} & -\dfrac{\partial \boldsymbol{Y}_1}{\partial \boldsymbol{y}_2} & \boldsymbol{0} \\ -\dfrac{\partial \boldsymbol{Y}_2}{\partial \boldsymbol{x}} & -\dfrac{\partial \boldsymbol{Y}_2}{\partial \boldsymbol{y}_1} & \boldsymbol{I} & \boldsymbol{0} \\ -\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}} & -\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}_1} & -\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}_2} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{I} \\ \dfrac{\mathrm{d}\boldsymbol{y}_1}{\mathrm{d}\boldsymbol{x}} \\ \dfrac{\mathrm{d}\boldsymbol{y}_2}{\mathrm{d}\boldsymbol{x}} \\ \dfrac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{I} \\ \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{0} \end{bmatrix}$$

(e) Coupled direct — functional approach

$$\begin{bmatrix} \boldsymbol{I} & -\left[\dfrac{\partial \boldsymbol{Y}_1}{\partial \boldsymbol{x}}\right]^T & -\left[\dfrac{\partial \boldsymbol{Y}_2}{\partial \boldsymbol{x}}\right]^T & -\left[\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right]^T \\ \boldsymbol{0} & \boldsymbol{I} & -\left[\dfrac{\partial \boldsymbol{Y}_2}{\partial \boldsymbol{y}_1}\right]^T & -\left[\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}_1}\right]^T \\ \boldsymbol{0} & -\left[\dfrac{\partial \boldsymbol{Y}_1}{\partial \boldsymbol{y}_2}\right]^T & \boldsymbol{I} & -\left[\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}_2}\right]^T \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \dfrac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} \\ \dfrac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{y}_1} \\ \dfrac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{y}_2} \\ \boldsymbol{I} \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{I} \end{bmatrix}$$

(f) Coupled adjoint — functional approach

$$\begin{bmatrix} \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ -\dfrac{\partial \boldsymbol{R}_1}{\partial \boldsymbol{x}} & -\dfrac{\partial \boldsymbol{R}_1}{\partial \boldsymbol{y}_1} & -\dfrac{\partial \boldsymbol{R}_1}{\partial \boldsymbol{y}_2} & \boldsymbol{0} \\ -\dfrac{\partial \boldsymbol{Y}_2}{\partial \boldsymbol{x}} & -\dfrac{\partial \boldsymbol{Y}_2}{\partial \boldsymbol{y}_1} & \boldsymbol{I} & \boldsymbol{0} \\ -\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}} & -\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}_1} & -\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}_2} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{I} \\ \dfrac{\mathrm{d}\boldsymbol{y}_1}{\mathrm{d}\boldsymbol{x}} \\ \dfrac{\mathrm{d}\boldsymbol{y}_2}{\mathrm{d}\boldsymbol{x}} \\ \dfrac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{I} \\ \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{0} \end{bmatrix}$$

(g) Hybrid direct

$$\begin{bmatrix} \boldsymbol{I} & -\left[\dfrac{\partial \boldsymbol{R}_1}{\partial \boldsymbol{x}}\right]^T & -\left[\dfrac{\partial \boldsymbol{Y}_2}{\partial \boldsymbol{x}}\right]^T & -\left[\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right]^T \\ \boldsymbol{0} & -\left[\dfrac{\partial \boldsymbol{R}_1}{\partial \boldsymbol{y}_1}\right]^T & -\left[\dfrac{\partial \boldsymbol{Y}_2}{\partial \boldsymbol{y}_1}\right]^T & -\left[\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}_1}\right]^T \\ \boldsymbol{0} & -\left[\dfrac{\partial \boldsymbol{R}_1}{\partial \boldsymbol{y}_2}\right]^T & \boldsymbol{I} & -\left[\dfrac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}_2}\right]^T \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \dfrac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{x}} \\ \dfrac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{r}_1} \\ \dfrac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{y}_2} \\ \boldsymbol{I} \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{I} \end{bmatrix}$$

(h) Hybrid adjoint

Figure 7.26: Derivation of the coupled direct (left column) and adjoint (right column) methods. The first row shows the original single system equations; in the second row we divide the residuals and state variables into two blocks; in the third row we present the functional approach to the coupled sensitivity equations; and the last row shows how the residual and functional approaches can be combined.

Coupled — Residual (Direct)

$$\begin{bmatrix} -\dfrac{\partial R_1}{\partial y_1} & -\dfrac{\partial R_1}{\partial y_2} \\ -\dfrac{\partial R_2}{\partial y_1} & -\dfrac{\partial R_2}{\partial y_2} \end{bmatrix} \begin{bmatrix} \dfrac{\mathrm{d}y_1}{\mathrm{d}x_1} & \dfrac{\mathrm{d}y_1}{\mathrm{d}x_2} \\ \dfrac{\mathrm{d}y_2}{\mathrm{d}x_1} & \dfrac{\mathrm{d}y_2}{\mathrm{d}x_2} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial R_1}{\partial x_1} & \dfrac{\partial R_1}{\partial x_2} \\ \dfrac{\partial R_2}{\partial x_1} & \dfrac{\partial R_2}{\partial x_2} \end{bmatrix}$$

$$\begin{bmatrix} -x_1 & -2 \\ 1 & -x_2^2 \end{bmatrix} \begin{bmatrix} \dfrac{\mathrm{d}y_1}{\mathrm{d}x_1} & \dfrac{\mathrm{d}y_1}{\mathrm{d}x_2} \\ \dfrac{\mathrm{d}y_2}{\mathrm{d}x_1} & \dfrac{\mathrm{d}y_2}{\mathrm{d}x_2} \end{bmatrix} = \begin{bmatrix} y_1 - \cos x_1 & 0 \\ 0 & 2x_2y_2 \end{bmatrix}$$

$$\frac{\mathrm{d}f_1}{\mathrm{d}x_1} = \frac{\partial F_1}{\partial x_1} + \frac{\partial F_1}{\partial y_1}\frac{\mathrm{d}y_1}{\mathrm{d}x_1} + \frac{\partial F_1}{\partial y_2}\frac{\mathrm{d}y_2}{\mathrm{d}x_1}$$

$$\frac{\mathrm{d}f_1}{\mathrm{d}x_1} = 0 + 1 \times \frac{\mathrm{d}y_1}{\mathrm{d}x_1} + 0 \times \frac{\mathrm{d}y_2}{\mathrm{d}x_1}$$

Coupled — Residual (Adjoint)

$$\begin{bmatrix} -\dfrac{\partial R_1}{\partial y_1} & -\dfrac{\partial R_2}{\partial y_1} \\ -\dfrac{\partial R_1}{\partial y_2} & -\dfrac{\partial R_2}{\partial y_2} \end{bmatrix} \begin{bmatrix} \dfrac{\mathrm{d}f_1}{\mathrm{d}r_1} & \dfrac{\mathrm{d}f_2}{\mathrm{d}r_1} \\ \dfrac{\mathrm{d}f_1}{\mathrm{d}r_2} & \dfrac{\mathrm{d}f_2}{\mathrm{d}r_2} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial F_1}{\partial y_1} & \dfrac{\partial F_2}{\partial y_1} \\ \dfrac{\partial F_1}{\partial y_2} & \dfrac{\partial F_2}{\partial y_2} \end{bmatrix}$$

$$\begin{bmatrix} -x_1 & 1 \\ -2 & -x_2^2 \end{bmatrix} \begin{bmatrix} \dfrac{\mathrm{d}f_1}{\mathrm{d}r_1} & \dfrac{\mathrm{d}f_2}{\mathrm{d}r_1} \\ \dfrac{\mathrm{d}f_1}{\mathrm{d}r_2} & \dfrac{\mathrm{d}f_2}{\mathrm{d}r_2} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & \sin x_1 \end{bmatrix}$$

$$\frac{\mathrm{d}f_1}{\mathrm{d}x_1} = \frac{\partial F_1}{\partial x_1} + \frac{\mathrm{d}f_1}{\mathrm{d}r_1}\frac{\partial R_1}{\partial x_1} + \frac{\mathrm{d}f_1}{\mathrm{d}r_2}\frac{\partial R_2}{\partial x_1}$$

$$\frac{\mathrm{d}f_1}{\mathrm{d}x_1} = 0 + \frac{\mathrm{d}f_1}{\mathrm{d}r_1}(y_1 - \cos x_1) + \frac{\mathrm{d}f_1}{\mathrm{d}r_2}0$$

Coupled — Functional (Direct)

$$\begin{bmatrix} 1 & -\dfrac{\partial Y_1}{\partial y_2} \\ -\dfrac{\partial Y_2}{\partial y_1} & 1 \end{bmatrix} \begin{bmatrix} \dfrac{\mathrm{d}y_1}{\mathrm{d}x_1} & \dfrac{\mathrm{d}y_1}{\mathrm{d}x_2} \\ \dfrac{\mathrm{d}y_2}{\mathrm{d}x_1} & \dfrac{\mathrm{d}y_2}{\mathrm{d}x_2} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial Y_1}{\partial x_1} & \dfrac{\partial Y_1}{\partial x_2} \\ \dfrac{\partial Y_2}{\partial x_1} & \dfrac{\partial Y_2}{\partial x_2} \end{bmatrix}$$

$$\begin{bmatrix} 1 & \frac{2}{x_1} \\ -\frac{1}{x_2^2} & 1 \end{bmatrix} \begin{bmatrix} \dfrac{\mathrm{d}y_1}{\mathrm{d}x_1} & \dfrac{\mathrm{d}y_1}{\mathrm{d}x_2} \\ \dfrac{\mathrm{d}y_2}{\mathrm{d}x_1} & \dfrac{\mathrm{d}y_2}{\mathrm{d}x_2} \end{bmatrix} = \begin{bmatrix} \frac{2y_2}{x_1^2} + \frac{\cos x_1}{x_1} - \frac{\sin x_1}{x_1^2} & 0 \\ 0 & -\frac{2y_1}{x_2^3} \end{bmatrix}$$

$$\frac{\mathrm{d}f_1}{\mathrm{d}x_1} = \frac{\partial F_1}{\partial x_1} + \frac{\partial F_1}{\partial y_1}\frac{\mathrm{d}y_1}{\mathrm{d}x_1} + \frac{\partial F_1}{\partial y_2}\frac{\mathrm{d}y_2}{\mathrm{d}x_1}$$

$$\frac{\mathrm{d}f_1}{\mathrm{d}x_1} = 0 + 1\frac{\mathrm{d}y_1}{\mathrm{d}x_1} + 0\frac{\mathrm{d}y_2}{\mathrm{d}x_1}$$

Coupled — Functional (Adjoint)

$$\begin{bmatrix} 1 & -\dfrac{\partial Y_2}{\partial y_1} \\ -\dfrac{\partial Y_1}{\partial y_2} & 1 \end{bmatrix} \begin{bmatrix} \dfrac{\mathrm{d}f_1}{\mathrm{d}y_1} & \dfrac{\mathrm{d}f_2}{\mathrm{d}y_1} \\ \dfrac{\mathrm{d}f_1}{\mathrm{d}y_2} & \dfrac{\mathrm{d}f_2}{\mathrm{d}y_2} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial F_1}{\partial y_1} & \dfrac{\partial F_2}{\partial y_1} \\ \dfrac{\partial F_1}{\partial y_2} & \dfrac{\partial F_2}{\partial y_2} \end{bmatrix}$$

$$\begin{bmatrix} 1 & -\frac{1}{x_2^2} \\ \frac{2}{x_1} & 1 \end{bmatrix} \begin{bmatrix} \dfrac{\mathrm{d}f_1}{\mathrm{d}y_1} & \dfrac{\mathrm{d}f_2}{\mathrm{d}y_1} \\ \dfrac{\mathrm{d}f_1}{\mathrm{d}y_2} & \dfrac{\mathrm{d}f_2}{\mathrm{d}y_2} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & \sin x_1 \end{bmatrix}$$

$$\frac{\mathrm{d}f_1}{\mathrm{d}x_1} = \frac{\partial F_1}{\partial x_1} + \frac{\mathrm{d}f_1}{\mathrm{d}y_1}\frac{\partial Y_1}{\partial x_1} + \frac{\mathrm{d}f_1}{\mathrm{d}y_2}\frac{\partial Y_2}{\partial x_1}$$

$$\frac{\mathrm{d}f_1}{\mathrm{d}x_1} = 0 + \frac{\mathrm{d}f_1}{\mathrm{d}y_1}\left(\frac{2y_2}{x_1^2} + \frac{\cos x_1}{x_1} - \frac{\sin x_1}{x_1^2}\right) + \frac{\mathrm{d}f_1}{\mathrm{d}y_2}0$$

Coupled — Hybrid (Direct)

$$\begin{bmatrix} -\dfrac{\partial R_1}{\partial y_1} & -\dfrac{\partial R_1}{\partial y_2} \\ -\dfrac{\partial Y_2}{\partial y_1} & 1 \end{bmatrix} \begin{bmatrix} \dfrac{\mathrm{d}y_1}{\mathrm{d}x_1} & \dfrac{\mathrm{d}y_1}{\mathrm{d}x_2} \\ \dfrac{\mathrm{d}y_2}{\mathrm{d}x_1} & \dfrac{\mathrm{d}y_2}{\mathrm{d}x_2} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial R_1}{\partial x_1} & \dfrac{\partial R_1}{\partial x_2} \\ \dfrac{\partial Y_2}{\partial x_1} & \dfrac{\partial Y_2}{\partial x_2} \end{bmatrix}$$

$$\begin{bmatrix} -x_1 & -2 \\ -\frac{1}{x_2^2} & 1 \end{bmatrix} \begin{bmatrix} \dfrac{\mathrm{d}y_1}{\mathrm{d}x_1} & \dfrac{\mathrm{d}y_1}{\mathrm{d}x_2} \\ \dfrac{\mathrm{d}y_2}{\mathrm{d}x_1} & \dfrac{\mathrm{d}y_2}{\mathrm{d}x_2} \end{bmatrix} = \begin{bmatrix} y_1 - \cos x_1 & 0 \\ 0 & -\frac{2y_1}{x_2^3} \end{bmatrix}$$

$$\frac{\mathrm{d}f_1}{\mathrm{d}x_1} = \frac{\partial F_1}{\partial x_1} + \frac{\partial F_1}{\partial y_1}\frac{\mathrm{d}y_1}{\mathrm{d}x_1} + \frac{\partial F_1}{\partial y_2}\frac{\mathrm{d}y_2}{\mathrm{d}x_1}$$

$$\frac{\mathrm{d}f_1}{\mathrm{d}x_1} = 0 + 1\frac{\mathrm{d}y_1}{\mathrm{d}x_1} + 0\frac{\mathrm{d}y_2}{\mathrm{d}x_1}$$

Coupled — Hybrid (Adjoint)

$$\begin{bmatrix} -\dfrac{\partial R_1}{\partial y_1} & -\dfrac{\partial Y_2}{\partial y_1} \\ -\dfrac{\partial R_1}{\partial y_2} & 1 \end{bmatrix} \begin{bmatrix} \dfrac{\mathrm{d}f_1}{\mathrm{d}r_1} & \dfrac{\mathrm{d}f_2}{\mathrm{d}r_1} \\ \dfrac{\mathrm{d}f_1}{\mathrm{d}y_2} & \dfrac{\mathrm{d}f_2}{\mathrm{d}y_2} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial F_1}{\partial y_1} & \dfrac{\partial F_2}{\partial y_1} \\ \dfrac{\partial F_1}{\partial y_2} & \dfrac{\partial F_2}{\partial y_2} \end{bmatrix}$$

$$\begin{bmatrix} -x_1 & -\frac{1}{x_2^2} \\ -2 & 1 \end{bmatrix} \begin{bmatrix} \dfrac{\mathrm{d}f_1}{\mathrm{d}r_1} & \dfrac{\mathrm{d}f_2}{\mathrm{d}r_1} \\ \dfrac{\mathrm{d}f_1}{\mathrm{d}y_2} & \dfrac{\mathrm{d}f_2}{\mathrm{d}y_2} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & \sin x_1 \end{bmatrix}$$

$$\frac{\mathrm{d}f_1}{\mathrm{d}x_1} = \frac{\partial F_1}{\partial x_1} + \frac{\mathrm{d}f_1}{\mathrm{d}r_1}\frac{\partial R_1}{\partial x_1} + \frac{\mathrm{d}f_1}{\mathrm{d}y_2}\frac{\partial Y_2}{\partial x_1}$$

$$\frac{\mathrm{d}f_1}{\mathrm{d}x_1} = 0 + \frac{\mathrm{d}f_1}{\mathrm{d}r_1}(y_1 - \cos x_1) + \frac{\mathrm{d}f_1}{\mathrm{d}y_2}0$$

Figure 7.27: Illustration of the coupled residual, functional, and hybrid approaches for the numerical example.

7.9 Concluding Remarks

In this chapter, we summarized and classified all known MDO architectures. We presented the architectures in a unified notation to draw comparisons between the architectures in order to contribute to a deeper and more encompassing understanding of the theory. We also introduced a classification of the architectures — summarized in Fig. 7.13 — that built on previous work to show how some architectures may be derived from others. Our hope is that this work will help point the way towards new MDO architectures and expand the scope of MDO theory and applications.

From a more practical perspective, much work remains in the area of benchmarking the existing architectures. The performance of the new architectures needs to be compared with that of the old architectures on a predefined set of test problems and the results should be independently corroborated. The development of MDO frameworks and test problem suites would be extremely useful.

Finally, new architectures for MDO are needed. A distributed architecture that exhibits fast convergence in most medium- and large-scale MDO problems may be thought of as the "holy grail" of MDO. The newest architectures in this chapter represent significant progress towards this objective. However, unless this fast convergence can be demonstrated for a wide range of problems, architecture development will remain an active area of research for the foreseeable future.

Bibliography

[1] Jeremy Agte, Olivier de Weck, Jaroslaw Sobieszczanski-Sobieski, Paul Arendsen, Alan Morris, and Martin Spieck. MDO: Assessment and direction for advancement — an opinion of one international group. *Structural and Multidisciplinary Optimization*, 40:17–33, 2010. doi:10.1007/s00158-009-0381-5.

[2] J Ahn and J H Kwon. An Efficient Strategy for Reliability-Based Multidisciplinary Design Optimization Using BLISS. *Structural and Multidisciplinary Optimization*, 31:363–372, 2006. doi:10.1007/s00158-005-0565-6.

[3] Natalia M Alexandrov and Robert Michael Lewis. Comparative Properties of Collaborative Optimization and Other Approaches to MDO. In *1st ASMO UK/ISSMO Conference on Engineering Design Optimization*, 1999.

[4] Natalia M Alexandrov and Robert Michael Lewis. Analytical and Computational Aspects of Collaborative Optimization for Multidisciplinary Design. *AIAA Journal*, 40(2):301–309, 2002. doi:10.2514/2.1646.

[5] Natalia M Alexandrov and Robert Michael Lewis. Reconfigurability in MDO Problem Synthesis, Part 1. In *10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Albany, NY, 2004.

[6] James T Allison, Brian Roth, Michael Kokkolaras, Ilan M Kroo, and Panos Y Papalambros. Aircraft Family Design Using Decomposition-Based Methods. In *11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, September 2006.

[7] James T Allison, David Walsh, Michael Kokkolaras, Panos Y Papalambros, and Matthew Cartmell. Analytical Target Cascading in Aircraft Design. In *44th AIAA Aerospace Sciences Meeting*, 2006.

[8] James T Allison, Michael Kokkolaras, and Panos Y Papalambros. On Selecting Single-Level Formulations for Complex System Design Optimization. *Journal of Mechanical Design*, 129: 898–906, 2007. doi:10.1115/1.2747632.

[9] R Balling and M R Rawlings. Collaborative Optimization with Disciplinary Conceptual Design. *Structural and Multidisciplinary Optimization*, 20(3):232–241, 2000. doi:10.1007/s001580050151.

[10] Richard J Balling and Jaroslaw Sobieszczanski-Sobieski. Optimization of Coupled Systems: A Critical Overview of Approaches. *AIAA Journal*, 34(1):6–17, 1996. doi:10.2514/3.13015.

[11] M S Bazaara, H D Sherali, and C M Shetty. *Nonlinear Programming: Theory and Algorithms*. John Wiley & Sons, 2006.

[12] J F Benders. Partitioning Procedures for Solving Mixed Variables Programming Problems. *Numerische Mathematik*, 4:238–252, 1962. doi:10.1007/BF01386316.

[13] Gal Berkooz, Philip Holmes, and John L Lumley. The Proper Orthogonal Decomposition in the Analysis of Turbulent Flows. *Annual Review of Fluid Mechanics*, 25:539–575, 1993. doi:10.1146/annurev.fl.25.010193.002543.

[14] Dimitri P Bertsekas. *Constrained Optimization and Lagrange Multiplier Methods*. Athena Scientific, 1996.

[15] Dimitri P Bertsekas and John N Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, 1997.

[16] Lorentz T Biegler, Omar Ghattas, Matthias Heinkenschloss, and Bart van Bloemen Waanders, editors. *Large-Scale PDE-Constrained Optimization*. Springer-Verlag, 2003.

[17] C. Bloebaum. Global sensitivity analysis in control-augmented structural synthesis. In *Proceedings of the 27th AIAA Aerospace Sciences Meeting*, Reno, NV, January 1989. AlAA 1989-0844.

[18] C L Bloebaum, P Hajela, and J Sobieszczanski-Sobieski. Non-Hierarchic System Decomposition in Structural Optimization. *Engineering Optimization*, 19(3):171–186, 1992. doi:10.1080/03052159208941227.

[19] C.L. Bloebaum. Coupling strength-based system reduction for complex engineering design. *Structural Optimization*, 10:113–121, 1995.

[20] C.L. Bloebaum, P. Hajela, and J. Sobieszczanski-Sobieski. Non-hierarchic system decomposition in structural optimization. In *Proceedings of the 3rd USAF/NASA Symposium on Recent Advances in Multidisciplinary Analysis and Optimization*, San Francisco, CA, 1990.

[21] V Y Blouin, G M Fadel, I U Haque, J R Wagner, and H B Samuels. Continuously Variable Transmission Design for Optimum Vehicle Performance by Analytical Target Cascading. *International Journal of Heavy Vehicle Systems*, 11:327–348, 2004. doi:10.1504/IJHVS.2004.005454.

[22] R D Braun, A A Moore, and I M Kroo. Collaborative Approach to Launch Vehicle Design. *Journal of Spacecraft and Rockets*, 34(4):478–486, 1997. doi:10.2514/2.3237.

[23] Robert D Braun. *Collaborative Optimization: An Architecture for Large-Scale Distributed Design*. PhD thesis, Stanford University, Stanford, CA, 1996.

[24] Robert D Braun and Ilan M Kroo. Development and Application of the Collaborative Optimization Architecture in a Multidisciplinary Design Environment. In N Alexandrov and M Y Hussaini, editors, *Multidisciplinary Design Optimization: State-of-the-Art*, pages 98–116. SIAM, 1997.

[25] Robert D Braun, Peter Gage, Ilan M Kroo, and Ian P Sobieski. Implementation and Performance Issues in Collaborative Optimization. In *6th AIAA, NASA, and ISSMO Symposium on Multidisciplinary Analysis and Optimization*, 1996.

[26] Nichols F Brown and John R Olds. Evaluation of Multidisciplinary Optimization Techniques Applied to a Reusable Launch Vehicle. *Journal of Spacecraft and Rockets*, 43(6):1289–1300, 2006. doi:10.2514/1.16577.

[27] Tyson R Browning. Applying the Design Structure Matrix to System Decomposition and Integration Problems: A Review and New Directions. *IEEE Transactions on Engineering Management*, 48(3):292–306, 2001. doi:10.1109/17.946528.

[28] Irene A Budianto and John R Olds. Design and Deployment of a Satellite Constellation Using Collaborative Optimization. *Journal of Spacecraft and Rockets*, 41(6):956–963, 2004. doi:10.2514/1.14254.

[29] Guobiao Cai, Jie Fang, Yuntao Zheng, Xiaoyan Tong, Jun Chen, and Jue Wang. Optimization of System Parameters for Liquid Rocket Engines with Gas-Generator Cycles. *Journal of Propulsion and Power*, 26(1):113–119, 2010. doi:10.2514/1.40649.

[30] Ting-Yu Chen. Calculation of the Move Limits for the Sequential Linear Programming Method. *International Journal for Numerical Methods in Engineering*, 36(15):2661–2679, 1993. doi:10.1002/nme.1620361510.

[31] Yong Chen, Xiaokai Chen, and Yi Lin. The Application of Analytical Target Cascading in Parallel Hybrid Electric Vehicle. In *IEEE Vehicle Power and Propulsion Conference*, pages 1602–1607, 2009. ISBN 9781424426010.

[32] Ian R Chittick and Joaquim R R A Martins. Aero-Structural Optimization Using Adjoint Coupled Post-Optimality Sensitivities. *Structural and Multidisciplinary Optimization*, 36: 59–70, 2008. doi:10.1007/s00158-007-0200-9.

[33] Ian R. Chittick and Joaquim R. R. A. Martins. An asymmetric suboptimization approach to aerostructural optimization. *Optimization and Engineering*, 10(1):133–152, March 2009. doi:10.1007/s11081-008-9046-2.

[34] R Choudhary, A Malkawi, and P Y Papalambros. Analytic Target Cascading in Simulation-Based Building Design. *Automation in Construction*, 14(4):551–568, 2005. doi:10.1016/j.autcon.2004.11.004.

[35] A J Conejo, F J Nogales, and F J Prieto. A Decomposition Procedure Based on Approximate Newton Directions. *Mathematical Programming*, 93:495–515, 2002. doi:10.1007/s10107-002-0304-3.

[36] Andrew R Conn, Nicholas I M Gould, and Philippe L Toint. *Trust Region Methods*. SIAM, Philadelphia, PA, 2000.

[37] Evin J Cramer, J E Dennis Jr., Paul D Frank, Robert Michael Lewis, and Gregory R Shubin. Problem Formulation for Multidisciplinary Optimization. *SIAM Journal on Optimization*, 4 (4):754–776, 1994. doi:10.1137/0804044.

[38] George B Dantzig and Phillip Wolfe. Decomposition Principle for Linear Programs. *Operations Research*, 8:101–111, 1960. doi:10.1287/opre.8.1.101.

[39] A J de Wit and F van Keulen. Numerical Comparison of Multilevel Optimization Techniques. In *Proceedings of the 48th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, April 2007.

[40] A J de Wit and F van Keulen. Overview of Methods for Multi-Level and/or Multi-Disciplinary Optimization. In *51st AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, Orlando, FL, April 2010.

[41] Angel-Victor DeMiguel. *Two Decomposition Algorithms for Nonconvex Optimization Problems with Global Variables*. PhD thesis, Stanford University, 2001.

[42] Angel-Victor DeMiguel and Walter Murray. An Analysis of Collaborative Optimization Methods. In *8th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis & Optimization*, Long Beach, CA, 2000.

[43] Victor DeMiguel and Walter Murray. A Local Convergence Analysis of Bilevel Decomposition Algorithms. *Optimization and Engineering*, 7(2):99–133, 2006. doi:10.1007/s11081-006-6835-3.

[44] R Enblom. Two-Level Numerical Optimization of Ride Comfort in Railway Vehicles. *Journal of Rail and Rapid Transit*, 220(1):1–11, 2006. doi:10.1243/095440905X33279.

[45] L F P Etman, M Kokkolaras, A T Hofkamp, P Y Papalambros, and J E Rooda. Coordination Specification in Distributed Optimal Design of Multilevel Systems Using the χ Language. *Structural and Multidisciplinary Optimization*, 29:198–212, 2005. doi:10.1007/s00158-004-0467-z.

[46] Anthony V Fiacco and Garth P McCormick. *Nonlinear Programming: Sequential Unconstrained Minimization Techniques*. SIAM, 1990.

[47] Ranjan Ganguli. Survey of recent developments in rotorcraft design optimization. *Journal of Aircraft*, 41(3):493–510, 2004.

[48] C Geethaikrishnan, P M Mujumdar, K Sudhakar, and V Adimurthy. A Hybrid MDO Architecture for Launch Vehicle Conceptual Design. In *51st AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, Orlando, FL, April 2010.

[49] Philip Geyer. Component-Oriented Decomposition for Multidisciplinary Design Optimization in Building Design. *Advanced Engineering Informatics*, 23(1):12–31, 2009. doi:10.1016/j.aei.2008.06.008.

[50] Philip E. Gill, Walter Murray, and Michael A. Saunders. SNOPT: An SQP algorithm for large-scale constrained optimization. *SIAM Review*, 47(1):99–131, 2005. doi:10.1137/S0036144504446096.

[51] Bryan Glaz, Peretz P Friedmann, and Li Liu. Helicopter Vibration Reduction Throughout the Entire Flight Envelope Using Surrogate-Based Optimization. *Journal of the American Helicopter Society*, 54:12007—-1–15, 2009. doi:10.4050/JAHS.54.012007.

[52] Justin Gray, Kenneth T Moore, and Bret A Naylor. OpenMDAO: An Open Source Framework for Multidisciplinary Analysis and Optimization. In *13th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Fort Worth, TX, September 2010.

[53] B Grossman, Z Gurdal, G J Strauch, W M Eppard, and R T Haftka. Integrated Aerodynamic/Structural Design of a Sailplane Wing. *Journal of Aircraft*, 25(9):855–860, 1988. doi:10.2514/3.45670.

[54] B. Grossman, R.T.Haftka, P.-J. Kao, D.M.Polen, and M.Rais-Rohani. Integrated aerodynamic-structural design of a transport wing. *Journal of Aircraft*, 27(12):1050–1056, 1990.

[55] Xiaoyu Gu, John E Renaud, Leah M Ashe, Stephen M Batill, Amrjit S Budhiraja, and Lee J Krajewski. Decision-Based Collaborative Optimization. *Journal of Mechanical Design*, 124 (1):1–13, 2002. doi:10.1115/1.1432991.

[56] Xiaoyu (Stacey) Gu, John E Renaud, and Charles L Penninger. Implicit Uncertainty Propagation for Robust Collaborative Optimization. *Journal of Mechanical Design*, 128(4):1001–1013, 2006. doi:10.1115/1.2205869.

[57] Raphael T Haftka. Automated Procedure for Design of Wing Structures to Satisfy Strength and Flutter Requirements. Technical Report TN D-7264, NASA Langley Research Center, Hampton, VA, 1973.

[58] Raphael T. Haftka. Optimization of flexible wing structures subject to strength and induced drag constraints. *AIAA Journal*, 14(8):1106–1977, 1977.

[59] Raphael T Haftka. Simultaneous Analysis and Design. *AIAA Journal*, 23(7):1099–1103, 1985. doi:10.2514/3.9043.

[60] Raphael T. Haftka and C. P. Shore. Approximate methods for combined thermal/structural design. Technical Report TP-1428, NASA, June 1979.

[61] Raphael T Haftka and Layne T Watson. Multidisciplinary Design Optimization with Quasiseparable Subsystems. *Optimization and Engineering*, 6:9–20, 2005. doi:10.1023/B:OPTE.0000048534.58121.93.

[62] Raphael T Haftka and Layne T Watson. Decomposition Theory for Multidisciplinary Design Optimization Problems with Mixed Integer Quasiseparable Subsystems. *Optimization and Engineering*, 7(2):135–149, 2006. doi:10.1007/s11081-006-6836-2.

[63] Jeongwoo Han and Panos Y Papalambros. A Sequential Linear Programming Coordination Algorithm for Analytical Target Cascading. *Journal of Mechanical Design*, 132(3):021003—-1–8, 2010. doi:10.1115/1.4000758.

[64] Jeongwoo Han and Panos Y Papalambros. A Note on the Convergence of Analytical Target Cascading with Infinite Norms. *Journal of Mechanical Design*, 132(3):034502—-1–6, 2010. doi:10.1115/1.4001001.

[65] Jeongwoo Han and Panos Y Papalambros. Optimal Design of Hybrid Electric Fuel Cell Vehicles Under Uncertainty and Enterprise Considerations. *Journal of Fuel Cell Science and Technology*, 7(2):021020—-1–9, 2010. doi:10.1115/1.3179762.

[66] Yuping He and John McPhee. Multidisciplinary Optimization of Multibody Systems with Application to the Design of Rail Vehicles. *Multibody System Dynamics*, 14(2):111–135, 2005. doi:10.1007/s11044-005-4310-0.

[67] Serhat Hosder, Layne T Watson, Bernard Grossman, William H Mason, Hongman Kim, Raphael T Haftka, and Steven E Cox. Polynomial Response Surface Approximations for the Multidisciplinary Design Optimization of a High Speed Civil Transport. *Optimization and Engineering*, 2:431–452, 2001. doi:10.1023/A:1016094522761.

[68] C.-H. Huang, J Galuski, and C L Bloebaum. Multi-Objective Pareto Concurrent Subspace Optimization for Multidisciplinary Design. *AIAA Journal*, 45(8):1894–1906, 2007. doi:10.2514/1.19972.

[69] George Q Huang and T. Qu. Extending Analytical Target Cascading for Optimal Configuration of Supply Chains with Alternative Autonomous Suppliers. *International Journal of Production Economics*, 115:39–54, 2008. doi:10.1016/j.ijpe.2008.04.008.

[70] George Q Huang, T Qu, David W L Cheung, and L Liang. Extensible Multi-Agent System for Optimal Design of Complex Systems Using Analytical Target Cascading. *International Journal of Advanced Manufacturing Technology*, 30:917–926, 2006. doi:10.1007/s00170-005-0064-3.

[71] Hong-Zhong Huang, Ye Tao, and Yu Liu. Multidisciplinary Collaborative Optimization Using Fuzzy Satisfaction Degree and Fuzzy Sufficiency Degree Model. *Soft Computing*, 12:995–1005, 2008. doi:10.1007/s00500-007-0268-6.

[72] K F Hulme and C L Bloebaum. A Simulation-Based Comparison of Multidisciplinary Design Optimization Solution Strategies Using CASCADE. *Structural and Multidisciplinary Optimization*, 19:17–35, 2000. doi:10.1007/s001580050083.

[73] Rajesh Kalavalapally, Ravi Penmetsa, and Ramana Grandhi. Multidisciplinary optimization of a lightweight torpedo structure subjected to an underwater explosion. *Finite Elements in Analysis and Design*, 43(2):103–111, December 2006. doi:10.1016/j.finel.2006.07.005.

[74] M Kaufman, V Balabanov, S L Burgee, A A Giunta, B Grossman, R T Haftka, W H Mason, and L T Watson. Variable-Complexity Response Surface Approximations for Wing Structural Weight in HSCT Design. *Computational Mechanics*, 18:112–126, 1996. doi:10.1007/BF00350530.

[75] Graeme J. Kennedy and Joaquim R. R. A. Martins. Parallel solution methods for aerostructural analysis and design optimization. In *Proceedings of the 13th AIAA/ISSMO Multidisciplinary Analysis Optimization Conference*, Forth Worth, TX, September 2010. AIAA 2010-9308.

[76] Graeme J Kennedy, Joaquim R R A Martins, and Jorn S Hansen. Aerostructural Optimization of Aircraft Structures Using Asymmetric Subspace Optimization. In *12th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Victoria, BC, Canada, 2008. AIAA.

[77] Gaetan Kenway, Graeme Kennedy, and Joaquim Martins. A scalable parallel approach for high-fidelity aerostructural analysis and optimization. In *53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*. American Institute of Aeronautics and Astronautics, Apr 2012. doi:10.2514/6.2012-1922. URL http://dx.doi.org/10.2514/6.2012-1922.

[78] H M Kim, M Kokkolaras, L S Louca, G J Delagrammatikas, N F Michelena, Z S Filipi, P Y Papalambros, J L Stein, and D N Assanis. Target Cascading in Vehicle Redesign: A Class VI Truck Study. *International Journal of Vehicle Design*, 29(3):199–225, 2002.

[79] Harrison M Kim, Wei Chen, and Margaret M Wiecek. Lagrangian Coordination for Enhancing the Convergence of Analytical Target Cascading. *AIAA Journal*, 44(10):2197–2207, 2006. doi:10.2514/1.15326.

[80] Hongman Kim, Scott Ragon, Grant Soremekun, Brett Malone, and Jaroslaw Sobieszczanski-Sobieski. Flexible Approximation Model Approach for Bi-Level Integrated System Synthesis. In *10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, pages 1–11, August 2004.

[81] Hyung Min Kim. *Target Cascading in Optimal System Design*. Ph.D. thesis, University of Michigan, 2001.

[82] Hyung Min Kim, Nestor F Michelena, Panos Y Papalambros, and Tao Jiang. Target Cascading in Optimal System Design. *Journal of Mechanical Design*, 125(3):474–480, 2003. doi:10.1115/1.1582501.

[83] Hyung Min Kim, D Geoff Rideout, Panos Y Papalambros, and Jeffrey L Stein. Analytical Target Cascading in Automotive Vehicle Design. *Journal of Mechanical Design*, 125(3):481–490, 2003. doi:10.1115/1.1586308.

[84] Srinivas Kodiyalam. Evaluation of Methods for Multidisciplinary Design Optimization (MDO), Phase I. Technical Report CR-1998-208716, NASA, September 1998.

[85] Srinivas Kodiyalam and Jaroslaw Sobieszczanski-Sobieski. Bilevel Integrated System Synthesis with Response Surfaces. *AIAA Journal*, 38(8):1479–1485, 2000. doi:10.2514/2.1126.

[86] Srinivas Kodiyalam and Jaroslaw Sobieszczanski-Sobieski. Multidisciplinary Design Optimization - Some Formal Methods, Framework Requirements, and Application to Vehicle Design. *International Journal of Vehicle Design*, 25:3–22, 2001. doi:10.1504/IJVD.2001.001904.

[87] Srinivas Kodiyalam and Charles Yuan. Evaluation of Methods for Multidisciplinary Design Optimization (MDO), Part 2. Technical Report CR-2000-210313, NASA, November 2000.

[88] Srinivas Kodiyalam, Mark Kremenetsky, and Stan Posey. Balanced HPC Infrastructure for CFD and Associated Multi-Discipline Simulations of Engineering Systems. *Journal of Aerospace Sciences and Technologies*, 61(3):434–443, 2009.

[89] M Kokkolaras, R Fellini, H M Kim, N F Michelena, and P Y Papalambros. Extension of the Target Cascading Formulation to the Design of Product Families. *Structural and Multidisciplinary Optimization*, 24:293–301, 2002. doi:10.1007/s00158-002-0240-0.

[90] M Kokkolaras, L S Louca, G J Delagrammatikas, N F Michelena, Z S Filipi, P Y Papalambros, J L Stein, and D N Assanis. Simulation-Based Optimal Design of Heavy Trucks by Model-Based Decomposition: An Extensive Analytical Target Cascading Case Study. *International Journal of Heavy Vehicle Systems*, 11:403–433, 2004. doi:10.1504/IJHVS.2004.005456.

[91] Michael Kokkolaras, Zissimous P Mourelatos, and Panos Y Papalambros. Design Optimization of Hierarchically Decomposed Multilevel Systems Under Uncertainty. *Journal of Mechanical Design*, 128(2):503–508, 2006. doi:10.1115/1.2168470.

[92] Ilan M Kroo. MDO for large-scale design. In N Alexandrov and M Y Hussaini, editors, *Multidisciplinary Design Optimization: State-of-the-Art*, pages 22–44. SIAM, 1997.

[93] Ilan M Kroo, Steve Altus, Robert Braun, Peter Gage, and Ian Sobieski. Multidisciplinary Optimization Methods for Aircraft Preliminary Design. In *5th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, 1994.

[94] Andrew B. Lambe and Joaquim R. R. A. Martins. Extensions to the design structure matrix for the description of multidisciplinary design, analysis, and optimization processes. *Structural and Multidisciplinary Optimization*, 2012. doi:10.1007/s00158-012-0763-y. (In press).

[95] Leon S Lasdon. *Optimization Theory for Large Systems*. The Macmillan Company, 1970.

[96] Laura A Ledsinger and John R Olds. Optimized Solutions for Kistler K-1 Branching Trajectories Using Multidisciplinary Design Optimization Techniques. *Journal of Spacecraft and Rockets*, 39(3):420–429, 2002. doi:10.2514/2.3825.

[97] Patrick A Legresley and Juan J Alonso. Improving the Performance of Design Decomposition Methods with POD. In *Proceedings of the 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, August 2004.

[98] Xiang Li, Weiji Li, and Chang'an Liu. Geometric Analysis of Collaborative Optimization. *Structural and Multidisciplinary Optimization*, 35:301–313, 2008. doi:10.1007/s00158-007-0127-1.

[99] Yanjing Li, Zhaosong Lu, and Jeremy J Michalek. Diagonal Quadratic Approximation for Parallelization of Analytical Target Cascading. *Journal of Mechanical Design*, 130(5):051402—-1–11, 2008. doi:10.1115/1.2838334.

[100] Zhijun Li, Michael Kokkolaras, Panos Y Papalambros, and S Jack Hu. Product and Process Tolerance Allocation in Multistation Compliant Assembly Using Analytical Target Cascading. *Journal of Mechanical Design*, 130(9):091701—-1–9, 2008. doi:10.1115/1.2943296.

[101] JiGuan G Lin. Analysis and Enhancement of Collaborative Optimization for Multidisciplinary Design. *AIAA Journal*, 42(2):348–360, 2004. doi:10.2514/1.9098.

[102] B Liu, R T Haftka, and L T Watson. Global-Local Structural Optimization Using Response Surfaces of Local Optimization Margins. *Structural and Multidisciplinary Optimization*, 27 (5):352–359, 2004. doi:10.1007/s00158-004-0393-0.

[103] Huibin Liu, Wei Chen, Michael Kokkolaras, Panos Y Papalambros, and Harrison M Kim. Probabilistic Analytical Target Cascading: A Moment Matching Formulation for Multilevel Optimization Under Uncertainty. *Journal of Mechanical Design*, 128(4):991–1000, 2006. doi:10.1115/1.2205870.

[104] E. Livne, L.A. Schmit, and P.P. Friedmann. Towards integrated multidisciplinary synthesis of actively controlled fiber composite wings. *Journal of Aircraft*, 27(12):979–992, December 1990. doi:10.2514/3.45972.

[105] Eli Livne. Integrated aeroservoelastic optimization: Status and direction. *Journal of Aircraft*, 36(1):122–145, 1999.

[106] Charles A. Mader, Joaquim R. R. A. Martins, Juan J. Alonso, and Edwin van der Weide. ADjoint: An approach for the rapid development of discrete adjoint solvers. *AIAA Journal*, 46(4):863–873, April 2008. doi:10.2514/1.29123.

[107] Valerie M Manning. *Large-Scale Design of Supersonic Aircraft via Collaborative Optimization*. PhD thesis, Stanford University, 1999.

[108] Christopher Marriage. *Automatic Implementation of Multidisciplinary Design Optimization Architectures Using πMDO*. Master's thesis, University of Toronto, 2008.

[109] Christopher J Marriage and Joaquim R R A Martins. Reconfigurable Semi-Analytic Sensitivity Methods and MDO Architectures within the πMDO Framework. In *Proceedings of the 12th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Victoria, BC, Canada, September 2008.

[110] Joaquim R. R. A. Martins, Peter Sturdza, and Juan J. Alonso. The complex-step derivative approximation. *ACM Transactions on Mathematical Software*, 29(3):245–262, 2003. doi:10.1145/838250.838251.

[111] Joaquim R. R. A. Martins, Juan J. Alonso, and James J. Reuther. High-fidelity aerostructural design optimization of a supersonic business jet. *Journal of Aircraft*, 41(3):523–530, 2004. doi:10.2514/1.11478.

[112] Joaquim R. R. A. Martins, Juan J. Alonso, and James J. Reuther. A coupled-adjoint sensitivity analysis method for high-fidelity aero-structural design. *Optimization and Engineering*, 6(1):33–62, March 2005. doi:10.1023/B:OPTE.0000048536.47956.62.

[113] Joaquim R R A Martins, Christopher Marriage, and Nathan Tedford. pyMDO: An Object-Oriented Framework for Multidisciplinary Design Optimization. *ACM Transactions on Mathematical Software*, 36(4):20:1–25, 2009. doi:10.1145/1555386.1555389.

[114] K. Maute, M. Nikbay, and C. Farhat. Coupled analytical sensitivity analysis and optimization of three-dimensional nonlinear aeroelastic systems. *AIAA Journal*, 39(11):2051–2061, 2001.

[115] C D McAllister, T W Simpson, K Hacker, K Lewis, and A Messac. Integrating Linear Physical Programming Within Collaborative Optimization for Multiobjective Multidisciplinary Design Optimization. *Structural and Multidisciplinary Optimization*, 29(3):178–189, 2005. doi:10.1007/s00158-004-0481-1.

[116] Charles D McAllister and Timothy W Simpson. Multidisciplinary Robust Design Optimization of an Internal Combustion Engine. *Journal of Mechanical Design*, 125(1):124–130, 2003. doi:10.1115/1.1543978.

[117] Jeremy J Michalek and Panos Y Papalambros. An Efficient Weighting Update Method to Achieve Acceptable Consistency Deviation in Analytical Target Cascading. *Journal of Mechanical Design*, 127:206–214, 2005. doi:10.1115/1.1830046.

[118] Jeremy J Michalek and Panos Y Papalambros. BB-ATC: Analytical Target Cascading Using Branch and Bound for Mixed Integer Nonlinear Programming. In *Proceedings of the ASME Design Engineering Technical Conference*, 2006.

[119] Jeremy J Michalek, Fred M Feinberg, and Panos Y Papalambros. Linking Marketing and Engineering Product Design Decisions via Analytical Target Cascading. *Journal of Product Innovation Management*, 22(1):42–62, 2005. doi:10.1111/j.0737-6782.2005.00102.x.

[120] Nestor F Michelena and Panos Y Papalambros. A Network Reliability Approach to Optimal Decomposition of Design Problems. *Journal of Mechanical Design*, 2:195–204, 1995. doi:10.1.1.35.147.

[121] Nestor F Michelena and Panos Y Papalambros. A Hypergraph Framework for Optimal Model-Based Decomposition of Design Problems. *Computational Optimization and Applications*, 8: 173–196, 1997. doi:10.1023/A:1008673321406.

[122] Nestor F Michelena, Hyungju Park, and Panos Y Papalambros. Convergence Properties of Analytical Target Cascading. *AIAA Journal*, 41(5):897–905, 2003. doi:10.2514/2.2025.

[123] Jorge Nocedal and Stephen J Wright. *Numerical Optimization*. Springer-Verlag, 2nd edition, 2006.

[124] Sharon L Padula and Ronnie E Gillian. Multidisciplinary Environments: A History of Engineering Framework Development. In *11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Porthsmouth, VA, September 2006.

[125] Sharon L Padula, Natalia Alexandrov, and Lawrence L Green. MDO Test Suite at NASA Langley Research Center. In *Proceedings of the 6th AIAA/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, Bellevue, WA, 1996.

[126] S Parashar and C L Bloebaum. Multi-Objective Genetic Algorithm Concurrent Subspace Optimization (MOGACSSO) for Multidisciplinary Design. In *11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, May 2006.

[127] S Parashar and C L Bloebaum. Robust Multi-Objective Genetic Algorithm Concurrent Subspace Optimization (R-MOGACSSO) for Multidisciplinary Design. In *11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, September 2006.

[128] Ruben E Perez. *A Multidisciplinary Optimization Framework for Flight Dynamics and Control Integration in Aircraft Design*. PhD thesis, University of Toronto, 2007.

[129] Ruben E Perez, Hugh H T Liu, and Kamran Behdinan. Evaluation of Multidisciplinary Optimization Approaches for Aircraft Conceptual Design. In *10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Albany, NY, August 2004.

[130] Ruben E Perez, Hugh H T Liu, and Kamran Behdinan. Multidisciplinary Optimization Framework for Control-Configuration Integration in Aircraft Conceptual Design. *Journal of Aircraft*, 43(6):1937–1948, 2006. doi:10.2514/1.22263.

[131] Daniele Peri and Emilio F. Campana. Multidisciplinary design optimization of a naval surface combatant. *Journal of Ship Research*, 47(1):1–12, 2003.

[132] Benjamin Potsaid, Yves Bellouard, and John Ting-Yung Wen. A Multidisciplinary Design and Optimization Methodology for the Adaptive Scanning Optical Microscope (ASOM). *Proceedings of the SPIE*, 6289:62890L1–12, 2006. doi:10.1117/12.680450.

[133] J E Renaud and G A Gabriele. Improved Coordination in Nonhierarchic System Optimization. *AIAA Journal*, 31(12):2367–2373, 1993. doi:10.2514/3.11938.

[134] J E Renaud and G A Gabriele. Approximation in Non-Hierarchic System Optimization. *AIAA Journal*, 32(1):198–205, 1994. doi:10.2514/3.11967.

[135] T D Robinson, K E Willcox, M S Eldred, and R Haimes. Multifidelity Optimization for Variable Complexity Design. In *11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, 2006.

[136] Brian Roth and Ilan Kroo. Enhanced Collaborative Optimization: Application to an Analytic Test Problem and Aircraft Design. In *12th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Victoria, British Columbia, Canada, September 2008.

[137] Brian D Roth. *Aircraft Family Design Using Enhanced Collaborative Optimization*. Ph.D. thesis, Stanford University, 2008.

[138] Andrzej Ruszczynski. On Convergence of an Augmented Lagrangian Decomposition Method for Sparse Convex Optimization. *Mathematics of Operations Research*, 20(3):634–656, 1995. doi:10.1287/moor.20.3.634.

[139] L A Schmit. Structural Design by Systematic Synthesis. In *2nd Conference on Electronic Computation*, pages 105–132, New York, NY, 1960. ASCE.

[140] L. A. Schmit, Jr. Structural synthesis — precursor and catalyst. recent experiences in multidisciplinary analysis and optimization. Technical Report CP-2337, NASA, 1984.

[141] Lucien A Schmit and William A Thornton. Synthesis of an Airfoil at Supersonic Mach Number. Technical Report CR 144, NASA, January 1965.

[142] L A Schmit Jr. and R K Ramanathan. Multilevel Approach to Minimum Weight Design Including Buckling Constraints. *AIAA Journal*, 16(2):97–104, 1978. doi:10.2514/3.60867.

[143] R S Sellar, S M Batill, and J E Renaud. Response Surface Based, Concurrent Subspace Optimization for Multidisciplinary System Design. In *34th AIAA Aerospace Sciences and Meeting Exhibit*, 1996.

[144] Jayashree Shankar, Calvin J Ribbens, Raphael T Haftka, and Layne T Watson. Computational Study of a Nonhierarchical Decomposition Algorithm. *Computational Optimization and Applications*, 2:273–293, 1993. doi:10.1007/BF01299452.

[145] M K Shin, B S Kang, and G J Park. Application of the Multidisciplinary Design Optimization Algorithm to the Design of a Belt-Integrated Seat While Considering Crashworthiness. *Journal of Automobile Engineering*, 219(11):1281–1292, 2005. doi:10.1243/095440705X34928.

[146] Moon-Kyun Shin and Gyung-Jin Park. Multidisciplinary Design Optimization Based on Independent Subspaces. *International Journal for Numerical Methods in Engineering*, 64: 599–617, 2005. doi:10.1002/nme.1380.

[147] Ian P Sobieski and Ilan M Kroo. Collaborative Optimization Using Response Surface Estimation. *AIAA Journal*, 38(10):1931–1938, 2000. doi:10.2514/2.847.

[148] J Sobieszczanski-Sobieski and R T Haftka. Multidisciplinary Aerospace Design Optimization: Survey of Recent Developments. *Structural Optimization*, 14(1):1–23, 1997. doi:10.1007/BF01197554.

[149] Jaroslaw Sobieszczanski-Sobieski. Optimization by Decomposition: A Step from Hierarchic to Non-Hierarchic Systems. Technical Report September, NASA Langley Research Center, Hampton, VA, 1988.

[150] Jaroslaw Sobieszczanski-Sobieski. Sensitivity of Complex, Internally Coupled Systems. *AIAA Journal*, 28(1):153–160, 1990. doi:10.2514/3.10366.

[151] Jaroslaw Sobieszczanski-Sobieski. Integrated System-of-Systems Synthesis. *AIAA Journal*, 46(5):1072–1080, 2008. doi:10.2514/1.27953.

[152] Jaroslaw Sobieszczanski-Sobieski. Sensitivity of complex, internally coupled systems. *AIAA Journal*, 28(1):153–160, 1990.

[153] Jaroslaw Sobieszczanski-Sobieski, Jeremy S Agte, and Robert R Sandusky Jr. Bilevel Integrated System Synthesis. *AIAA Journal*, 38(1):164–172, 2000. doi:10.1.1.35.4601.

[154] Jaroslaw Sobieszczanski-Sobieski, Troy D Altus, Matthew Phillips, and Robert R Sandusky Jr. Bilevel Integrated System Synthesis for Concurrent and Distributed Processing. *AIAA Journal*, 41(10):1996–2003, 2003. doi:10.2514/2.1889.

[155] D V Steward. The Design Structure Matrix: A Method for Managing the Design of Complex Systems. *IEEE Transactions on Engineering Management*, 28:71–74, 1981.

[156] Jun Tajima, Fujio Momiyama, and Naohiro Yuhara. A New Solution for Two-Bag Air Suspension System with Leaf Spring for Heavy-Duty Vehicle. *Vehicle System Dynamics*, 44(2): 107–138, 2006. doi:10.1080/00423110500385907.

[157] R V Tappeta and J E Renaud. Multiobjective Collaborative Optimization. *Journal of Mechanical Design*, 119:403–411, 1997. doi:10.1115/1.2826362.

[158] R V Tappeta, S Nagendra, and J E Renaud. A Multidisciplinary Design Optimization Approach for High Temperature Aircraft Engine Components. *Structural Optimization*, 18(2-3): 134–145, 1999. doi:10.1007/BF01195988.

[159] Nathan P Tedford and Joaquim R R A Martins. Benchmarking Multidisciplinary Design Optimization Algorithms. *Optimization and Engineering*, 11:159–183, 2010. doi:10.1007/s11081-009-9082-6.

[160] S Tosserams, L F P Etman, P Y Papalambros, and J E Rooda. An augmented lagrangian relaxation for analytical target cascading using the alternating direction method of multipliers. *Structural and Multidisciplinary Optimization*, 31(3):176–189, 2006. doi:10.1007/s00158-005-0579-0.

[161] S Tosserams, L F P Etman, and J E Rooda. An Augmented Lagrangian Decomposition Method for Quasiseparable Problems in MDO. *Structural and Multidisciplinary Optimization*, 34:211–227, 2007. doi:10.1007/s00158-006-0077-z.

[162] S Tosserams, L F P Etman, and J E Rooda. Augmented lagrangian coordination for distributed optimal design in MDO. *International Journal for Numerical Methods in Engineering*, 73:1885–1910, 2008. doi:10.1002/nme.2158.

[163] S Tosserams, L F P Etman, and J E Rooda. A Classification of Methods for Distributed System Optimization based on Formulation Structure. *Structural and Multidisciplinary Optimization*, 39(5):503–517, 2009. doi:10.1007/s00158-008-0347-z.

[164] S Tosserams, L F P Etman, and J E Rooda. Block-Separable Linking Constraints in Augmented Lagrangian Coordination in MDO. *Structural and Multidisciplinary Optimization*, 37 (5):521–527, 2009. doi:10.1007/s00158-008-0244-5.

[165] S Tosserams, L F P Etman, and J E Rooda. Multi-Modality in Augmented Lagrangian Coordination for Distributed Optimal Design. *Structural and Multidisciplinary Optimization*, 40:329–352, 2010. doi:10.1007/s00158-009-0371-7.

[166] S Tosserams, L F P Etman, and J E Rooda. A Micro-Accelerometer MDO Benchmark Problem. *Structural and Multidisciplinary Optimization*, 41(2):255–275, 2010. doi:10.1007/s00158-009-0422-0.

[167] S Tosserams, M Kokkolaras, L F P Etman, and J E Rooda. A Nonhierarchical Formulation of Analytical Target Cascading. *Journal of Mechanical Design*, 132(5):051002—-1–13, 2010. doi:10.1115/1.4001346.

[168] G N Vanderplaats. An Efficient Feasible Directions Algorithm for Design Synthesis. *AIAA Journal*, 22(11):1633–1640, 1984. doi:10.2514/3.8829.

[169] Stephen J Wright. *Primal-Dual Interior Point Methods*. SIAM, 1997.

[170] Brett A Wujek, John E Renaud, Stephen M Batill, and Jay B Brockman. Concurrent Subspace Optimization Using Design Variable Sharing in a Distributed Computing Environment. *Concurrent Engineering: Research and Applications*, 4(4):361–377, 1996. doi:10.1177/1063293X9600400405.

[171] S I Yi, J K Shin, and G J Park. Comparison of MDO Methods with Mathematical Examples. *Structural and Multidisciplinary Optimization*, 39:391–402, 2008. doi:10.1007/s00158-007-0150-2.

[172] Parviz M Zadeh and Vassili V Toropov. Multi-Fidelity Multidisciplinary Design Optimization Based on Collaborative Optimization Framework. In *9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, Atlanta, GA, September 2002.

[173] Parviz M Zadeh, Vassili V Toropov, and Alastair S Wood. Metamodel-Based Collaborative Optimization Framework. *Structural and Multidisciplinary Optimization*, 38:103–115, 2009. doi:10.1007/s00158-008-0286-8.

[174] Ke-Shi Zhang, Zhong-Hua Han, Wei-Ji Li, and Wen-Ping Song. Bilevel Adaptive Weighted Sum Method for Multidisciplinary Multi-Objective Optimization. *AIAA Journal*, 46(10): 2611–2622, 2008. doi:10.2514/1.36853.

Chapter 8

Optimization Under Uncertainty

8.1 Introduction

Uncertainty is always present in engineering design. Manufacturing processes create deviations from the designed model, operating conditions vary from the ideal, and some parameters are inherently variable. Optimization with deterministic inputs can lead to poorly performing designs. To create robust and reliable designs we must treat the relevant parameters and design variables as random variables. **Optimization under uncertainity** (OUU) is the optimization of systems in the presence of uncertain parameters or design variables.

We have seen a simple example of this already. Figure 1.6 in the first chapter shows an airfoil optimized for a deterministic Mach number. The result is an airfoil with low drag at Mach 0.74 (as requested!). However, any deviation from the target Mach number causes significant drag penalties. In other words, the design is not robust. The design is improved by considering the inherent variability in cruise Mach number (in a very basic way). In this case, the Mach number is treated as a random variable with four values of equal probability Fig. 1.7. Even a change this simple significantly increased the robustness of the design. The minimum objective value (drag) is not as low, but it is less sensitive to deviations from the desired operating point. As is always true, a trade-off in peak performance is required to achieve robustness.

We call a design **robust** if its performance is less sensitive to inherent variability. In other words, the *objective* function is less sensitive to variations in the random design variables and parameters. Similarly, we call a design **reliable** if it is less prone to failure under variability. In other words, the *constraints* have a lower probability of being violated under variation in the random design variables and parameters.

In this chapter we will discuss how uncertainty can be used in the objective function allowing for robust designs, and how it can be used in constraints allowing for reliable designs. We will also discuss a few different methods for uncertainty propagation. Uncertainty quantification, and optimization under uncertainty, is a deep and growing field. This chapter only presents a brief introduction to optimization under uncertainty.

8.2 Statistics Review

This section presents a very brief review of some aspects of statistics and probability theory.

Imagine measuring the axial strength of a rod by perform a tensile test with many rods, each designed to be identical. Even with "identical" rods, every time you perform the test you get a different answer (hopefully only slightly differently). This variation has many potential sources

include variation in the manufactured size and shape, in the composition of the material, in the contact between the rod and testing fixture, etc. In this example, we would call the axial strength a *random variable*, and the result from one test would be a random sample. The random variable axial strength is a function of several other random variables like: bar length, bar diameter, material Young's modulus, etc.

One measurement doesn't give us much information, but if we perform the test many times we can learn a lot about the axial strength. For example, we can estimate the mean value of the axial strength. In general, we can estimate the mean of some variable x that is measured N times as:

$$\mu_x = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{8.1}$$

Note that this is actually a sample mean, which would differ from the population mean (the true mean if you could measure every bar). With enough samples the sample mean will approach the population mean. In this brief introduction we won't distinguish between sample and population statistics.

Another important quantity is the variance or standard deviation. This is a measure of spread. The unbiased estimate of the variance is:

$$\sigma_x^2 = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu_x)^2 \tag{8.2}$$

and the standard deviation is just the square root of the variance. A small variance implies that measurements are clustered tightly around the mean, whereas a large variance means that measurements are spread out far from the mean. The variance can also be written in the equivalent, but more computationally friendly format:

$$\sigma_x^2 = \frac{1}{N-1}\left(\sum_{i=1}^{N} x_i^2 - N\mu_x^2\right) \tag{8.3}$$

More generally, we might want to know what the probability is of getting a bar with a specific axial strength. In our testing, we could tabulate the frequency of each measurement in a histogram. If done enough times, it would define a smooth curve Fig. 8.1. This curve is called the *probability density function* (PDF), $p(x)$, and it tells us the probability of a certain value occurring.

$$\text{Prob}[x = a] = p(a) \tag{8.4}$$

It also gives the probability of getting a value with a certain range:

$$\text{Prob}[a \leq x \leq b] = \int_a^b p(x)dx \tag{8.5}$$

Clearly the total integral of the PDF must be one.

$$\int_{-\infty}^{\infty} p(x)dx = 1 \tag{8.6}$$

From the PDF we can also measure various statistics like the mean:

$$\mu_x = E[x] = \int_{-\infty}^{\infty} xp(x)dx \tag{8.7}$$

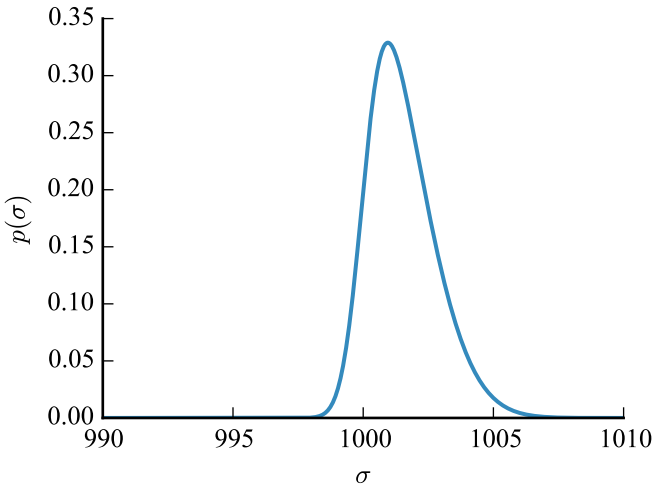


Figure 8.1: Probability distribution function for axial strength of rod.

This quantity is also referred to as the expected value of x ($E[x]$).

A PDF can be discrete, but often we fit or use a functional form for the PDF. One of the most popular distributions is the Gaussian or Normal distribution. Its PDF is:

$$p(x;\mu,\sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} \exp \frac{-(x-\mu)^2}{2\sigma^2} \tag{8.8}$$

For a Gaussian distribution the mean and variance are clearly visible in the function, but keep in mind these quantities are defined for any distribution. Figure 8.2 shows two normal distributions with different means and standard deviations. A few other popular distribution are shown in Fig. 8.3, and many others exist as well.

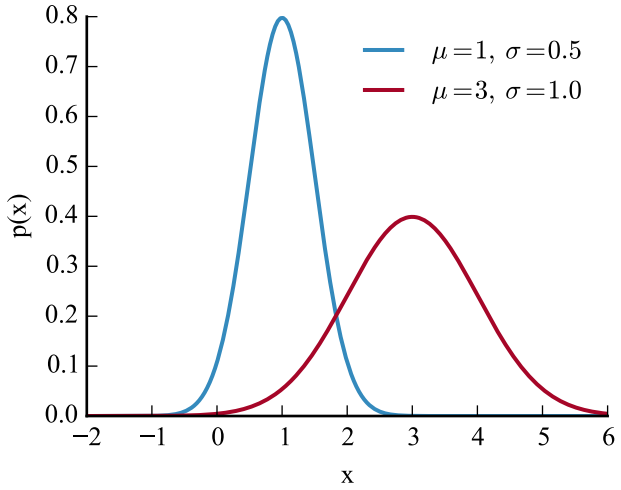
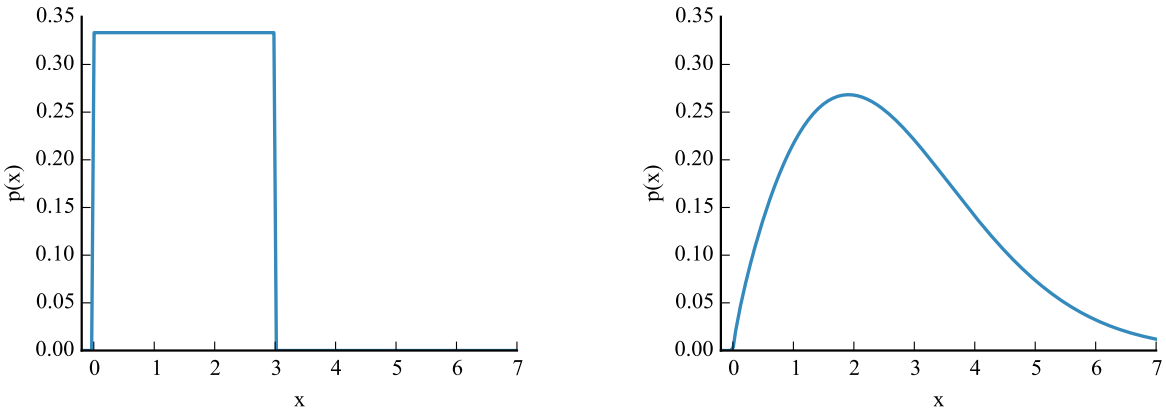
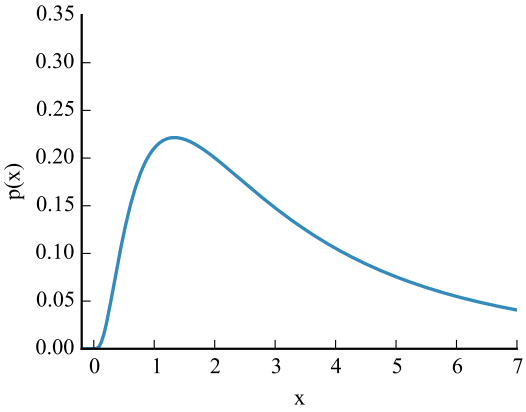


Figure 8.2: Two normal distributions. Changing the mean causes a shift along the x-axis. Increasing the standard deviation causes the PDF to spread out.

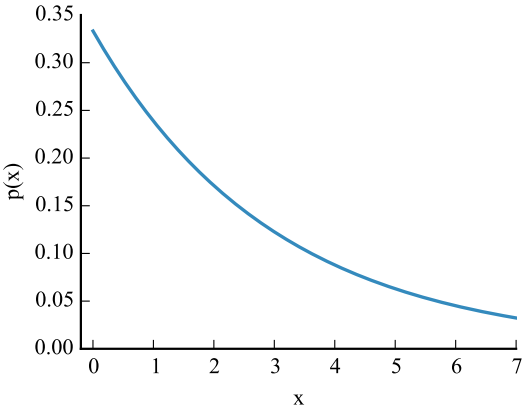


(a) Uniform distribution

(b) Weibull distribution



(c) Lognormal distribution



(d) Exponential distribution

Figure 8.3: A few other example probability distributions.

8.3 Robust Design

How does uncertainty affect the robustness of a design? Consider the simple motivating example shown in Fig. 8.4. From observation, we can see that the global minimum is located at $x = 0.9$ and has a magnitude of about -3.4. The methods we have discussed in previous chapters can be used to find this deterministic optimum.

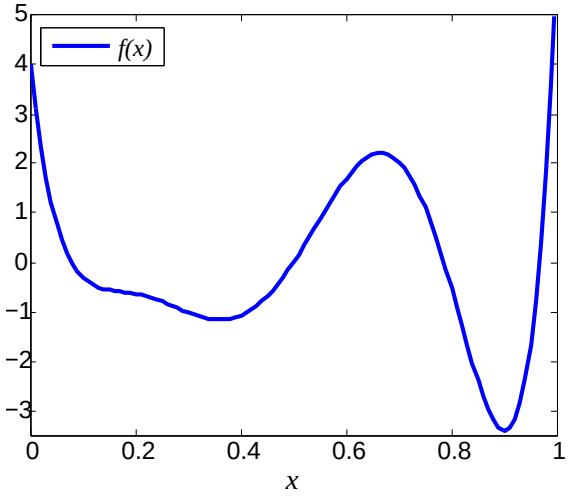


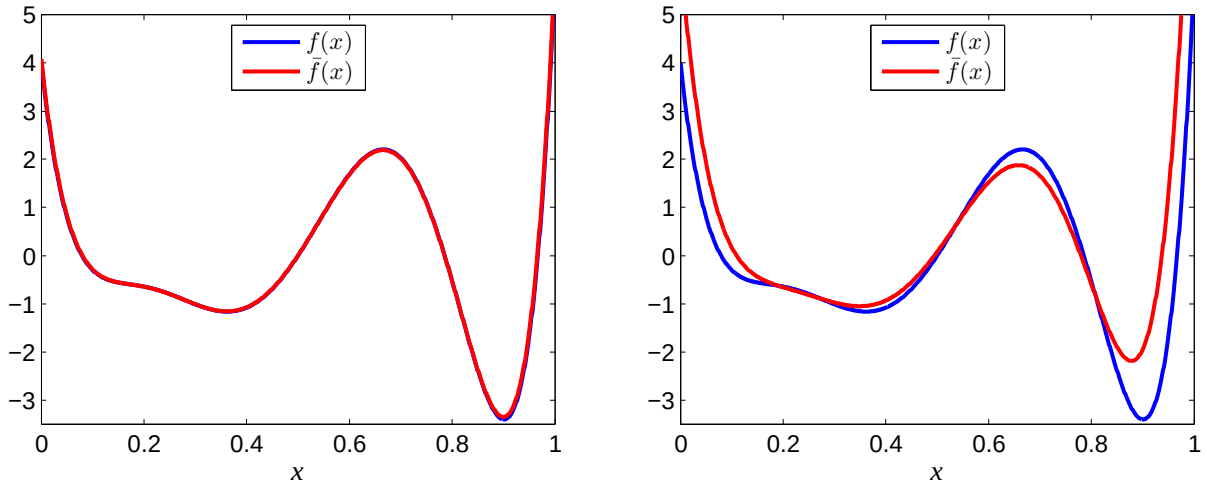
Figure 8.4: Output f as a function of deterministic input x.

Now consider an alternative scenario. What if x is not a deterministic variable, but rather a normally distributed random variable (e.g., an output from a manufacturing process)? Based on our previous discussion we can intuit what to expect. The deterministic optimum has the best performance, but is in a steep valley and will suffer significant performance losses if the input has inherent variability. Depending on how large the variability is, the performance will degrade further. By contrast, the performance of a point near $x = 0.25$ will be less affected by variation. If the variability is high enough, then the optimal point may be closer to $x = 0.25$ than $x = 0.9$.

Figure 8.5 demonstrates this quantitatively. In each figure, the deterministic function output is shown in comparison to the mean of the stochastic function output. For each plot, the standard deviation of the normally distributed input is changed. Notice, that as the variability increases in Figs. 8.5a to 8.5c the optimal location changes slightly and the magnitude of the objective increases, and by Fig. 8.5d the variability is large enough that the location of the optimum has completely changed. Comparatively, the performance of the design near $x = 0.25$ has changed much less.

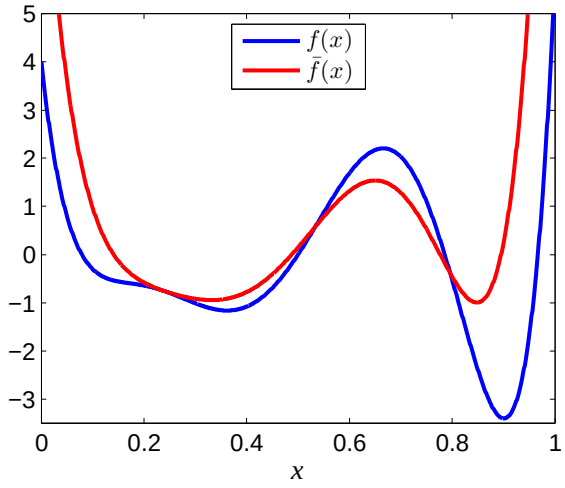
From this simple example we can see that the optimal design, in the presence of variability, can be quite different from the optimal design found using deterministic inputs. Thus, ignoring input uncertainty in our design processes can be costly. This example also illustrates the danger of promising designs that are found in a "deep valley". In the presence of variability these so-called point designs will suffer significant performance losses. This is just like the airfoil problem, where the initial design performed very well at a single Mach number, but paid a steep penalty in off-design conditions. Often, we are more interested in designs that fall in a "broad valley" like the second location near $x = 0.3$ in the analytic example, or the airfoil design that included variability in its objective. These designs are robust; they are less sensitive to variability.

In this example our output was a random variable, and we used its mean value as our objective function. However, there are other useful forms of OUU objective functions that may be more

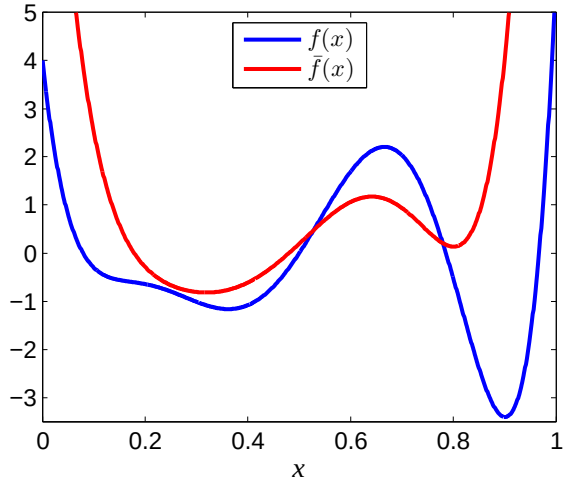


(a) $\sigma = 0.01$

(b) $\sigma = 0.05$



(c) $\sigma = 0.075$



(d) $\sigma = 0.1$

Figure 8.5: Blue curve shows deterministic output. Red curve shows stochastic output based on a normally distributed input variable with standard deviation as specified.

suitable. Consider the following options:

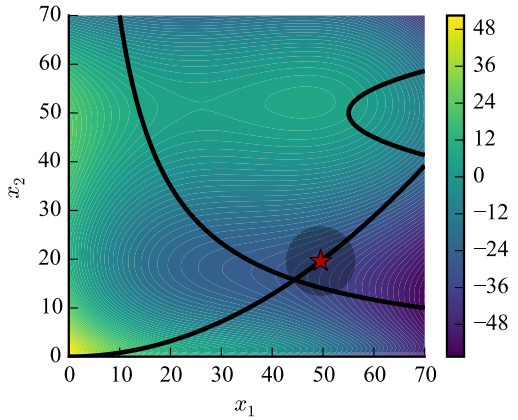
1. Minimize the mean of the function: $\mu_f(x)$.
2. Minimize the standard deviation (or variance) of the function: $\sigma_f(x)$.
3. Perform a multiobjective optimization trading off the the mean and standard deviation. Or, minimize the convex sum of the mean and standard deviation: $\alpha\mu_f(x) + (1-\alpha)\sigma_f(x)$ for some scalar $\alpha \in [0, 1]$.
4. Minimize a reliability metric (see next section for further discussion on reliability): Prob$[f(x) > f_{crit}]$, which means to minimize the probability that the objective exceeds some critical value.

Option 3 acknowledges the multiobjective nature of OUU: there is a trade-off between the objective of performance (μ_f) and its variability (σ_f). A Pareto front can be a useful tool to assess trade-offs in these two potential objectives.

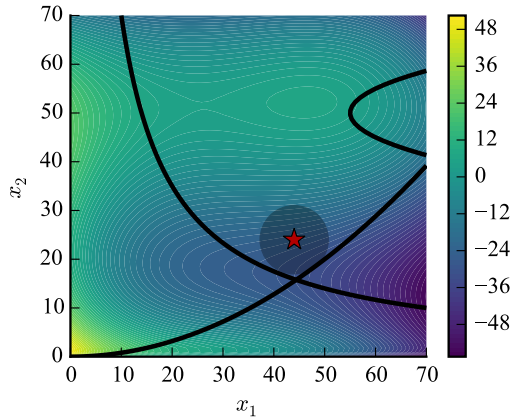
8.4 Reliability

In addition to affecting the objective function, uncertainty also affects constraints. Consider the optimal point in the Barnes function Fig. 8.6a. With deterministic inputs, the optimal value sits right on the constraint. An uncertainty ellipse shown around the optimal point highlights the fact that the deterministic optimum is not reliable. Any variability in the inputs will create a significant probability for one or more of the constraints to be violated.

Conversely, Fig. 8.6b shows a reliable optimum where it is highly probable that the design will satisfy all constraints under the input variation. Increased reliability presents a performance trade-off with a corresponding increase in the objective function.



(a) The constrained deterministic optimum is denoted by the star, the feasible region is the region just above the star.



(b) A reliable optimal point is denoted by the star.

Figure 8.6: Contour plots of Barnes function, the three lines are the three constraints of the problem. An uncertainty ellipse is shown around the optimal points.

In some engineering disciplines, increasing the reliability is handled in a basic way through safety factors (which are deterministic, but usually derived through statistical means). In other

words, if we were constraining the stress (σ) in a beam to be less than the material's yield stress (σ_y) we would not want to use a constraint of the form:

$$\sigma(x) \leq \sigma_y \tag{8.9}$$

This would be dangerous because we know there is inherent variability in the loads, and uncertainty in the yield stress of the material. Instead we often use a simple safety factor.

$$\sigma(x) \leq \eta\sigma_y \tag{8.10}$$

where η is a total safety factor including partial safety factors from loads, materials, failure modes, etc. Of course, not all applications have standards-driven safety factors already determined. The statistical approach discussed in this chapter is useful in these situations to allow for reliable designs.

To create reliable design we change our deterministic inequality constraints:

$$c(x) \leq 0 \tag{8.11}$$

to the form:

$$\text{Prob}[c(x) \leq 0] \geq R \tag{8.12}$$

where R is the reliability level. In words, we want the probability of constraint satisfaction to exceed some pre-selected reliability level. For example, if we set $R = 0.999$ the solution must satisfy the constraints with a probability of 99.9%. Thus, we can explicitly set the reliability level that we wish to achieve (with of course trade-offs in the level of performance for the objective function).

Up to this point we have not discussed how we will compute the output statistics: μ_f, σ_f, $\text{Prob}[c(x) \leq 0]$. This is the subject of the following section.

8.5 Forward Propagation

In the previous sections we have assumed that we know the statistics (mean, standard deviation, etc.) of the *outputs* of interest (objectives and constraints). However, generally we do not have that information. Instead, we only know the probability density functions of the *inputs*. We need a way to efficiently propagate these random input variables through a potentially expense computational model to find the statistics of the outputs of interest.

Forward propagation methods propagate input uncertainties through a numerical model to compute output uncertainties. We will explore two methods for forward propagation: Monte Carlo Simulation and Stochastic Collocation. Many other advanced methods exist, but these are introduced for their relative simplicity.

These two methods are both **nonintrusive**. This means that we just need to be able to run the deterministic model repeatedly, in other words, the simulation tool can be treated as a black box. Conversely, **intrusive** methods require changes inside the model to compute stochastic outputs. We are familiar with these two categories from Chapter 4 on computing derivatives. Methods like finite differencing are nonintrusive. This makes them easy to use, but less efficient and accurate. Intrusive methods like the adjoint method are accurate and efficient, but require a lot more upfront effort. Similar benefits exist for intrusive uncertainty propagation methods, however discussion of these methods is out of the scope of our brief introduction.

8.5.1 Monte Carlo Simulation

The most widely known and simplest forward propagation method is Monte Carlo Simulation (MCS). The basic idea with Monte Carlo is that the output probability distribution can be approximated by running the simulation many times with inputs sampled from the input probability distributions. The basic steps are:

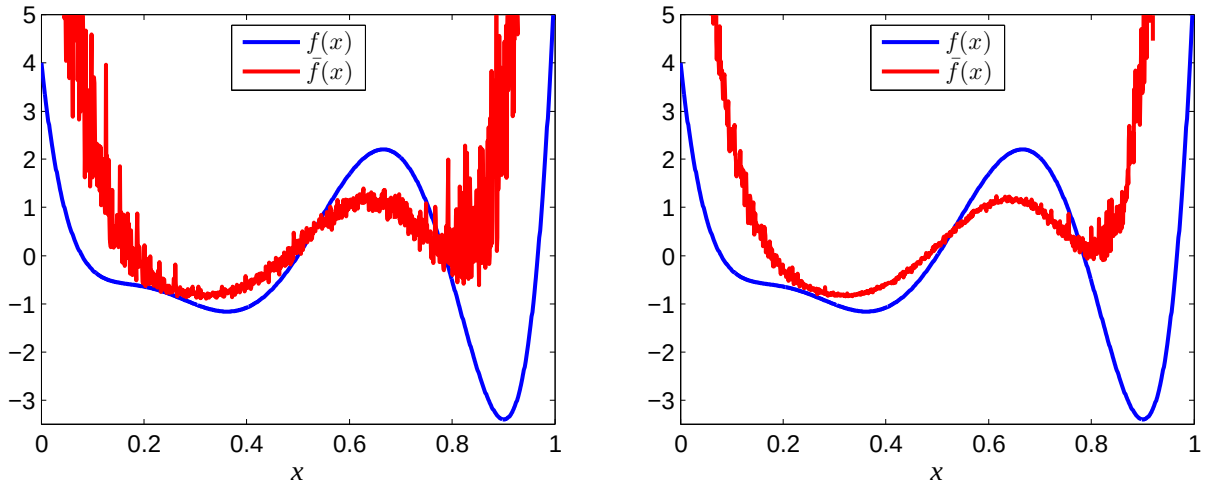
1. Sample N points x_i from the input probability distribution functions.
2. Evaluate the outputs at these points: $f_i = f(x_i)$.
3. Compute statistics on the discrete output distribution f_i (using Eqs. (8.1) and (8.3)).

We can also estimate Prob$[c(x) \leq 0]$ just by counting how many times the constraint was satisfied and dividing by N. If we evaluate enough samples, our output statistics will converge to the true values by virtual of the central limit theorem.

MSC has two major advantages. The first advantage is that the algorithm is easy to parallelize. All of the function evaluations are completely independent. The second advantage is that the convergence rate is independent of the dimension of the stochastic input space. In other words, whether we have 3 or 300 random input variables, the convergence rate will be similar. The major disadvantage is that the convergence rate is slow: $\mathcal{O}(1/\sqrt{N})$. Thus, to improve accuracy by one decimal place requires approximately 100 *times* more samples.

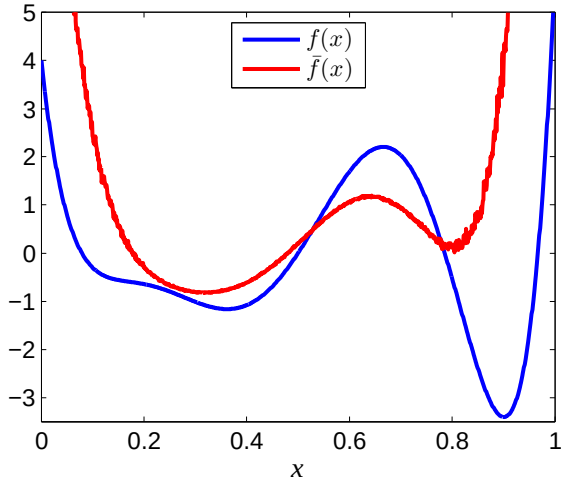
Consider the example shown previously in Fig. 8.5d. The red curve in that figure was computed using a forward propagation method. We can generate that curve using MCS. Figure 8.7 shows the mean of the output function using MCS for 100, 1,000, 10,000 and 100,000 samples. We see that a very large number of samples is required to achieve well converged statistics.

One approach to achieve converged statistics with fewer iterations is to use Latin Hypercube Sampling (LHS). LHS allows one to better approximate the input distributions with fewer samples, and is discussed further in Chapter 9 on surrogate-based optimization. Even with better sampling methods, MCS still requires lots of simulations, which can be particularly prohibitive if used as part of an optimization under uncertainty problem.

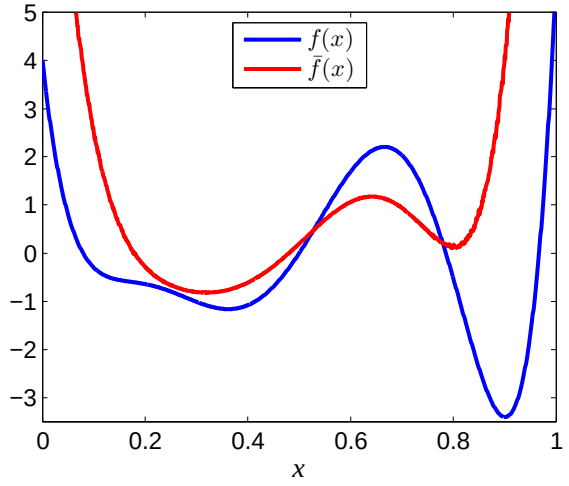


(a) $N = 100$

(b) $N = 1,000$



(c) $N = 10,000$



(d) $N = 100,000$

Figure 8.7: The blue curve shows deterministic output. The red curve shows stochastic output from MCS using the specified number of samples (N) on a normally distributed input variable.

8.5.2 Stochastic Collocation

Monte Carlo is easy to implement and use, but because of the large number of required iterations, it is only practical for very inexpensive output functions. Stochastic Collocation (SC) offers an alternative for certain probability density functions [1]. Its major advantage is that if the stochastic input dimension is small then it is very efficient. Its primary disadvantage is that it suffers from the curse of dimensionality (i.e., it is not efficient at large stochastic dimensions) If the input space is small, Stochastic Collocation is preferable. If the input space is large, Monte Carlo is preferable.

The basic idea for SC is to use advanced numerical integration methods to estimate the output statistics. This requires fewer function calls than would be required from random sampling. There are several different numerical integration methods that you are likely already familiar with: midpoint rule, trapezoidal integration, Simpson's method. Generally, these methods require a large number of points in order to produce accurate values for the integrand. An alternative approach involves fitting a polynomial of degree $N-1$ using N points. If we carefully choose the points to evaluate at, and the weights for each point, we can achieve a very accurate estimate of the integral with a relatively small number of evaluation points (assuming the output function is continuously differentiable). Mathematically this is expressed as:

$$\int_a^b f(x)dx = \sum_{i=1}^{\infty} w_i f(x_i) \approx \sum_{i=1}^{N} w_i f(x_i) \tag{8.13}$$

and is known as Gaussian quadrature. The basis functions f are chosen carefully to be orthogonal to the input probability distribution. Table 8.1 lists some of the distributions and the corresponding polynomials that we can address using this method.

Table 8.1: Polynomial basis functions in the Askey family used to integrate output statistics for different input probability distributions.

Distribution	Polynomials
Uniform	Legendre
Gaussian	Hermite
Exponential	Laguerre
Beta	Jacobi

The basic steps for SC are:

1. Sample f at specially chosen quadrature points, x_i, defined for a given probability density function.

2. Compute statistics from a weighted sum. For example the mean of the function can be estimated as

$$\mu_f \approx \sum_{i=1}^{N} w_i f(x_i) \tag{8.14}$$

The details of the methodology are beyond the scope of this introductory overview, but as an example we show the same problem shown previously, but now estimate the mean of f with stochastic collocation (Fig. 8.8). Only 6 function calls were required, as compared to 100,000 with MCS to achieve similar convergence.

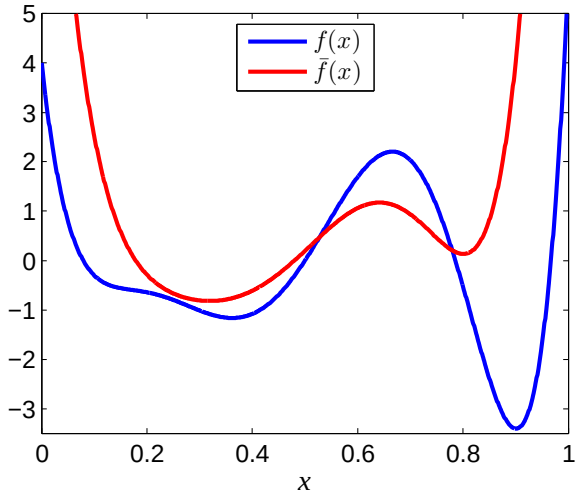


Figure 8.8: The blue curve shows deterministic output. The red curve shows stochastic output from stochastic collocation using $N = 6$ on a normally distributed input variable.

Extensions to higher dimensions are straightforward, but suffer from the curse of dimensionality. The number of evaluations scales with N^d where d is the number of dimensions. However, rather than using full tensor grids, sparse grid methods have been shown to reduce the computational burden while still retaining reasonable accuracy [3]. While a great improvement, this does not fully eliminate the scaling challenge, and thus this methodology is only appropriate for a modest number of random variables and parameters.

8.6 Approximate Reliability Methods

Integrated OUU can be computationally intensive and require significant expertise. While recent advances in forward propagation methods (particular quadrature methods and intrusive spectral methods) have made such problems much more tractable, some approximation methods are still popular in conceptual design applications because of their simplicity and small computational burden. Two such methods are proposed by Parkinson, Sorensen, and Pourhassan [2].

These approximation methods are used for reliability constraints, in other words, approximating the constraint: $\text{Prob}[c(x) \leq 0] > R$. They are not used to estimate output statistics for μ_f and σ_f. Only an overview of the methods is mentioned here, for further details refer to the referenced article.

8.6.1 Worst-Case Tolerances

Consider a constraint with random design variable and random parameters:

$$c(x, p) \geq 0 \tag{8.15}$$

To simplify further, the random variables x and p need not be described with probability distributions, but only with tolerances (i.e., the half-width of a uniform probability distribution). We consider the worst-case scenario where every variation is assumed to decrease the constraint margin.

To first-order, the *worst-case* variation in the constraint is:

$$\Delta c = \sum_{i=1}^{n} \left| \frac{\partial c}{\partial x_i} \Delta x_i \right| + \sum_{j=1}^{m} \left| \frac{\partial c}{\partial p_j} \Delta p_j \right| \tag{8.16}$$

where n is the number of design variables, m the number of parameters, Δx_i is the tolerance of the i^{th} design variable, and Δp_j is the tolerance of the j^{th} parameter. Once we compute a worst-case variation for each constraint, we adjust each constraint to the more reliable:

$$c - \Delta c \geq 0 \tag{8.17}$$

The procedure is to:

1. Compute the deterministic optimum
2. Estimate the worst-case variation Δc
3. Adjust the constraint to $c - \Delta c \geq 0$ and reoptimize

Note that this process is very simple as we only need to make an analytic estimate of uncertainty, and only need to do it once. However, we have assumed that the robust optimum is close enough to the deterministic optimum so that a first-order approximation is sufficiently accurate, and that a constant value for the variation is reasonable. Additionally, we assume that the worst-case scenario is not overly conservative. Despite the approximations, the simplicity of the method is appealing and is much more effective than just using a deterministic optimum.

8.6.2 Transmitted Variance

If we do have an input probability distribution (instead of just tolerances) we can use a better, but similarly simple approach. Like the previous method, we assume a first-order approximation, but this time of the transmitted variance:

$$\sigma_c^2 = \sum_{i=1}^{n} \left(\frac{\partial c}{\partial x_i} \sigma_{x_i} \right)^2 + \sum_{j=1}^{m} \left(\frac{\partial c}{\partial p_j} \sigma_{p_j} \right)^2 \tag{8.18}$$

Then we adjust the constraint as:

$$c(x) - k\sigma_c \geq 0 \tag{8.19}$$

where k is chosen for a desired reliability level R. For example, $k = 2$ implies a reliability level of 97.72% (one-sided tail of the normal distribution). Thus, this approximates the $\text{Prob}[c(x) \leq 0] > R$ constraint, but with the assumption that the output distribution is normally distributed. In many cases an output distribution is reasonably approximately as normal, but for cases with nonnormal output this method can introduce large error.

The methodology is similar:

1. Compute the deterministic optimum
2. Estimate the transmitted variance σ_c
3. Adjust the constraint to $c(x) - k\sigma_c \geq 0$ for some desired reliability level and reoptimize

With multiple active constraints, one must be careful to appropriately chose the reliability level for each constraint such that the overall reliability is in the desired range. Often the simplifying assumption is made that the constraints are uncorrelated and thus the total reliability is the product of the reliabilities of each constraint.

For both methods, if we have exact gradients using the methods discussed in Chapter 4, then it can be effective to use the methods iteratively in the optimization rather than only once after the first optimization. Doing so relaxes the assumption that the robust optimum is near the deterministic optimum. This is only feasible with exact derivatives not just because of computationally cost, but also because providing gradients of the new constraints means that we need second derivatives of c. If those are numerically estimated they are likely to be highly inaccurate and lead to poor convergence. For this reason, the simpler optimize, estimate, reoptimize method is often used.

While these methods are approximate, they are easy to use and the magnitude of error is usually quite appropriate for the conceptual design phase.

Bibliography

[1] Ivo Babuška, Fabio Nobile, and Raúl Tempone. A stochastic collocation method for elliptic partial differential equations with random input data. *SIAM Journal on Numerical Analysis*, 45(3): 1005–1034, jan 2007. doi:10.1137/050645142. URL http://dx.doi.org/10.1137/050645142.

[2] Alan Parkinson, Carl Sorensen, and Nader Pourhassan. A general approach for robust optimal design. *Journal of Mechanical Design*, 115(1):74, 1993. doi:10.1115/1.2919328. URL http://dx.doi.org/10.1115/1.2919328.

[3] Sergey A Smolyak. Quadrature and interpolation formulas for tensor products of certain classes of functions. *Dokl. Akad. Nauk SSSR*, 4(123):240–243, 1963.