Constrained Functions of N Variables: Non-Gradient Based Methods

Gerhard Venter

Stellenbosch University
Outline

• Outline
• Constrained Optimization
• Non-gradient based methods
• Genetic Algorithms (GA)
• Particle Swarm Optimization (PSO)
• Parallelization
Course Outline

• Until now
  – Chapter 1: Basic concepts
  – Chapter 2: Functions of one variable
  – Chapter 3: Unconstrained functions of N variables
  – Chapter 4: Linear programming
  – Chapter 5: Sequential unconstrained minimization techniques

• Today: Constrained functions of N variables
  – Chapter 6: Non-gradient based optimization
Constrained Optimization

• Standard Form
  – Find the set of design variables that will:

Minimize: \( F(X) \)

Subject To:
\( g_j(X) \leq 0 \quad j = 1, m \)
\( h_k(X) = 0 \quad k = 1, l \)
\( X_i^L \leq X_i \leq X_i^U \quad i = 1, n \)
Kuhn Tucker Conditions

- Kuhn-Tucker conditions provide necessary conditions for a (local) optimum

\[ X^* \text{ is feasible} \]
\[ \lambda_j g_j (X^*) = 0 \]
\[ \nabla F (X^*) + \sum_{j=1}^{m} \lambda_j \nabla g_j (X^*) + \sum_{k=1}^{l} \lambda_{m+k} \nabla h_k (X^*) = 0 \]
\[ \lambda_j \geq 0 \quad j = 1, m \]
\[ \lambda_{m+k} \text{ unrestricted in sign} \]
Non-Gradient Based Methods

- Simplest is a random search
- Easy to implement
- Very robust
- Very inefficient
- Improve random search by adding some logic
  - Example DOE search with move-limits
  - Referred to as a structured random search
- We will consider two structured random searches in this class
Non-Gradient Based Methods

• Non-gradient based optimization algorithms have gained a lot of attention recently
  – Easy to program
  – Global properties
  – Require no gradient information
    – *High computational cost*
    – *Tuned for each problem*

• Typically based on some physical phenomena
  – Genetic algorithms
  – Simulated annealing
  – Particle swarm optimization
Non-Gradient Based Methods

- Can be classified as a structured random search
- Does not move from one design point to the next - makes use of a population of design points
- Numerically robust
- Increased changes of finding global or near global optimum designs
- Provides a number of good designs instead of a single optimum
Genetic Algorithms

- Derived from biology and making use of Darwin’s principal of survival of the fittest
- First developed in 1975 by Holland
- Population adapts to its underlying environment through evolution
  - Characteristics stored in chromosomal strings
  - Structured but random exchange of genetic information
  - Better designs have increased changes of reproducing
Genetic Operators

- Start with initial population
- Reproduce by using
  - Selection
  - Cross-over
  - Occasional mutation
  - Elitist strategy
- Design variables are encoded in bit-strings that are equivalent to chromosomal strings
Design Variable Encoding

- Traditionally binary numbers were used
- Example

\[
F(X) \quad X = \{X_1, X_2, X_3, X_4\}
\]

\[
0 \leq X_1, X_4 \leq 15
\]

\[
0 \leq X_2 \leq 7
\]

\[
0 \leq X_3 \leq 3
\]

\[
\begin{bmatrix}
0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1
\end{bmatrix}
\]

\[
X_1 = 6 \quad X_2 = 5 \quad X_3 = 3 \quad X_4 = 11
\]
Binary Encoding

- Mainly used for historical reasons
  - Number of bits required - \( m \)
  
  \[ 2^m \geq \frac{X^u - X^l}{X_{incr}} + 1 \]

  - Use to represent integer and real numbers
  - Smallest possible alphabet
  - Added complexity that is not needed

- Can use actual values (integer or real) in our bit string
Real Number Encoding

- Laminate stacking sequence optimization
  - Layers can only be stacked using $0^\circ, 45^\circ, -45^\circ, 90^\circ$
  - Bit string for 16-ply laminate $[\pm 45^\circ/0_4/90_2]^s$
Real Number Encoding

• Full string – $4^{16} = 4.3 \times 10^9$ combinations
  
  \[
  [2 \ 3 \ 1 \ 1 \ 1 \ 1 \ 4 \ 4 \ 4 \ 4 \ 1 \ 1 \ 1 \ 1 \ 3 \ 2]
  \]

• Use symmetry – $4^8 = 65,536$ combinations
  
  \[
  [2 \ 3 \ 1 \ 1 \ 1 \ 1 \ 4 \ 4]
  \]

• Balanced with stack variables – $3^4 = 81$ combinations
  
  \[
  [2 \ 1 \ 1 \ 3]
  \]
Typical Genetic Algorithm

Start

Create Initial Swarm

Perform analyses

Converged

Yes

End

No

Selection

Cross-over

Mutation

Elitist Strategy
Initial Population

• Initial population
  – Fixed size population
  – Fixed chromosome length
  – No fixed rule for determining population size
  – Randomly distributed throughout the design space
Selection and Fitness

• Selection and fitness
  – Selection gives more fit designs higher chance of passing their genes to the next generation
  – Fitness based on the objective function
  – Use SUMT approach for constrained problems

• Selection
  – Using roulette wheel approach

\[ r_i = \frac{F_i}{\sum_{j=1}^{n_s} F_j} \]
Selection

- Roulette wheel based on objective values
  - All objective function values must be positive
  - Little differentiation between design at end of optimization
- Use roulette wheel based on rank
  - Can take positive and negative objective function values
  - Better differentiation between designs

\[ r_i = \frac{2(n_s - i + 1)}{(n_s + 1)n_s} \]
Apply Selection

- Create roulette wheel with \( n_s \) segments
- Create random number between 0 and 1
- Determine segment on roulette wheel that contains the random number
- Segment ID corresponds to design ID
- Repeat to obtain two parents for reproduction through cross-over
Cross-over

- Select two parents for reproduction
- Apply probability of cross-over, $\text{pc}$
  - $\text{pc}$ is typically close to 1
- Mix parent genes using cross-over to obtain one (or two) children
- One point cross-over

<table>
<thead>
<tr>
<th>Parent 1</th>
<th>82</th>
<th>43</th>
<th>17</th>
<th>11</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent 2</td>
<td>85</td>
<td>31</td>
<td>62</td>
<td>29</td>
<td>66</td>
</tr>
<tr>
<td>Child</td>
<td>85</td>
<td>31</td>
<td>17</td>
<td>11</td>
<td>8</td>
</tr>
</tbody>
</table>
## Cross-over

- **Two point cross-over**

<table>
<thead>
<tr>
<th></th>
<th>82</th>
<th>43</th>
<th>17</th>
<th>11</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parent 2</td>
<td>85</td>
<td>31</td>
<td>62</td>
<td>29</td>
<td>66</td>
</tr>
<tr>
<td>Child</td>
<td>82</td>
<td>43</td>
<td>62</td>
<td>29</td>
<td>8</td>
</tr>
</tbody>
</table>

- **Uniform cross-over**

<table>
<thead>
<tr>
<th></th>
<th>82</th>
<th>43</th>
<th>17</th>
<th>11</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parent 2</td>
<td>85</td>
<td>31</td>
<td>62</td>
<td>29</td>
<td>66</td>
</tr>
<tr>
<td>Random</td>
<td>0.71</td>
<td>0.13</td>
<td>0.01</td>
<td>0.58</td>
<td>0.30</td>
</tr>
<tr>
<td>Child</td>
<td>85</td>
<td>43</td>
<td>17</td>
<td>29</td>
<td>8</td>
</tr>
</tbody>
</table>
Cross-over

- One point averaging cross-over
  - Especially useful for real-numbers
  - Generate random number between 1 and the length of the string
  - Say we generate 3.2
  - Elements 1 and 2 from Parent 1, elements 4 and 5 from parent 2, element 3 from

\[ C^3 = 0.2P^3_1 + 0.8P^3_2 \]

<table>
<thead>
<tr>
<th>Parent 1</th>
<th>82</th>
<th>43</th>
<th>17</th>
<th>11</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent 2</td>
<td>85</td>
<td>31</td>
<td>62</td>
<td>29</td>
<td>66</td>
</tr>
<tr>
<td>Child</td>
<td>85</td>
<td>31</td>
<td>53</td>
<td>11</td>
<td>8</td>
</tr>
</tbody>
</table>
Mutation

- Prevent premature loss of genetic information by introducing randomness in the population
- Important to create information not contained in initial population
- Apply probability of performing mutation, $p_m$
  - $p_m$ is typically small
- Randomly select gene to mutate
- Randomly modify gene
Cost/Reliability

- Algorithm is very costly
- Cost is determined by
  - Problem size – number of design variables and number of combinations
  - Constraint definitions
- Reliability can be improved by
  - Performing more generations
  - Increase the population size
  - Repeat the optimization run
Typical Applications

- Analysis is cheap to perform
- Problems that contain many local minima
  - Not that common in engineering applications
- Problems that has no gradient information
  - Selecting different cross-sections for a truss structure
- Combinatorial type problems
  - Composite laminate design – find optimum thickness and ply orientation
  - D-optimal design of experiments
Particle Swarm Optimization

- Particle Swarm Optimization (PSO) is a fairly recent addition to the family of non-gradient based optimization algorithms.
- PSO is based on a simplified social model that is closely tied to swarming theory.
  - Example is a swarm of bees searching for a food source.
  - Use knowledge of individual particles and swarm as a whole.
- **Basic Assumption:** A population of individuals adapts by returning to promising regions found previously.
Algorithm Overview

Initial Swarm

Final Swarm

- **Particle** $P_i$ properties:
  - Position: $(A_{1j}, A_{2j})$
  - Velocity Vector
  - Objective: $W(A_{1j}, A_{2j})$
  - Constraints: $S(A_{1j}, A_{2j}, P_1, P_2)$
How is PSO Implemented

Position Update:

\[ x_{k+1}^i = x_k^i + v_{k+1}^i \Delta t \]

Velocity Vector:

\[ v_{k+1}^i = w v_k^i + c_1 r_1 \frac{p_i^i - x_k^i}{\Delta t} + c_2 r_2 \frac{p_g^g - x_k^i}{\Delta t} \]

- Inertia
- Trust in self
- Trust in swarm
Enhancements

• Convergence criterion
  – Monitor the maximum change in the objective function for a specified number of consecutive design iterations

• Integer/Discrete problems
  – Unlike genetic algorithms, PSO is inherently a continuous algorithm
  – Round the new position to closest value before performing function evaluations

• Craziness
  – Similar to mutation operator in Genetic Algorithms
  – Prevent premature convergence
• **Constrained problems**
  
  – Combine constraints with objective: exterior penalty function:
    \[
    \tilde{F}(X, r_p) = F(X) + r_p P(X)
    \]

  – Deal with violated design points (including points outside bounds) based on usable, feasible directions idea

  – Create new velocity vector to point back to the feasible design space by re-setting the velocity vector:
    \[
    \mathbf{v}_{k+1}^i = c_1 r_1 \frac{p_k^i - x_k^i}{\Delta t} + c_2 r_2 \frac{p_k^g - x_k^i}{\Delta t}
    \]
Start

Create Initial Swarm

Perform analyses

Converged

Yes ➔ End

No ➔ Adjust Parameters

New Velocity Vector

Update Position

Apply Craziness
Example Problem 1

- **Global optimum:**
  - Point = (0,0)
  - Obj = 0

- **Local optima:**
  - Point = (-1.75, -0.87)
  - Obj = 0.3
  - Point = (1.75, 0.87)
  - Obj = 0.3
Example Problem 2

- Global optimum:
  - Point = (0,0)
  - Obj = 1000

- Many local optima
Example Problem

**Multidisciplinary design optimization of a transport aircraft wing structure**

- System level optimization
  - *PSO Algorithm*
- Structural sub-optimization
  - *GENESIS*
- Aerodynamic analysis
  - *Simplified analysis*
System Level Optimization

- Unconstrained problem
- **Maximize**: Range
- **Design Variables**:  
  - Aspect ratio  
  - Depth-to-chord ratio  
  - Number of internal spars  
  - Number of internal ribs  
  - Wing cover construction
Structural Sub-Optimization

- Structural analysis using a simplified FE model of the wing box
- Objective is to minimize the weight of the wing
- Both stress and local buckling constraints are applied

80 Design Variables

206 Design Variables
• Assumed pressure distribution is converted to nodal loads for the structural sub-optimization

• **Simplified drag analysis:** Drag of the current wing is calculated based on the total drag of the reference wing

• Range is calculated from the Breguet formula
  – Constant Take-Off Gross Weight
  – Structural weight saving is converted to fuel weight

• Wing is build to a jig shape to compensate for the aerodynamic-structure interaction
System Level Analysis

- Update FE Model
- Apply Pressure Distribution
- Structural Sub-Optimization
- Calculate Aerodynamic Drag
- Calculate Range
Repeated optimization ten times, each with different initial swarm
<table>
<thead>
<tr>
<th></th>
<th>Reference Wing</th>
<th>Optimum Wing (Sandwich)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>5000.0 n. mi.</td>
<td>5375 n. mi.</td>
</tr>
<tr>
<td>A</td>
<td>6.8571</td>
<td>9.2360</td>
</tr>
<tr>
<td>h/c</td>
<td>0.12</td>
<td>0.11</td>
</tr>
</tbody>
</table>
What about using a gradient-based optimizer to solve this problem?

- Look at all possible combinations for the three truly discrete variables
- For each combination solve a two design variable continuous optimization problem
- Results in 32 optimization problems
- Total of 526 analyses (1500 for the PSO)
- Parallel implementation implications
Gradient-Based Results

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Reference Wing</th>
<th>DOT Optimum</th>
<th>PSO Optimum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aspect Ratio</td>
<td>6.8571</td>
<td>6.9100</td>
<td>9.2360</td>
</tr>
<tr>
<td>Depth-to-Chord Ratio</td>
<td>0.01</td>
<td>0.08</td>
<td>0.11</td>
</tr>
<tr>
<td>Num. Internal Spars</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Num. Internal Ribs</td>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Construction</td>
<td>–</td>
<td>Sandwich</td>
<td>Sandwich</td>
</tr>
<tr>
<td>Range (n. mi.)</td>
<td>5000</td>
<td>5225</td>
<td>5375</td>
</tr>
<tr>
<td>Number of Analyses</td>
<td>–</td>
<td>526</td>
<td>1500</td>
</tr>
</tbody>
</table>
Numerical Noise

Don’t forget about
NUMERICAL NOISE

- Severe numerical noise due to the structural sub-optimization
  - Noise due to different levels of convergence
  - Iterative process
Remarks

- PSO algorithm was able to produce consistent results in the presence of truly discrete design variables and severe numerical noise.
- PSO computational cost is high, but using a gradient based optimizer is not a feasible alternative:
  - Numerical noise
  - Implementation issues
  - Larger number of discrete combinations
- PSO has significant potential for massively parallel processing.
Simplified Flowchart

1. Start
2. Create Initial Swarm
3. Perform analyses
   - Converged
     - Yes: End
     - No: Adjust Parameters
   - New Velocity Vector
4. Update Position
5. Apply Craziness
Parallel PSO: Synchronized

Wait for all analyses to complete before starting next design iteration

- What has been done so far
- Straight forward to implement
- Overall logic of the algorithm is not changed
  - produce same results
- Poor parallel speedup
  - Some processors remain idle part-time!
Parallel PSO: Asynchronized

Goal is to make all processors work all of the time

- Do not wait for the current design iteration to complete before starting the next
  - Update all point information as soon as it becomes available
  - Update iteration information as soon as a design iteration is completed
- Overall logic of algorithm is changed – may produce different results
- Very good parallel speedup
Parallel Environment

- Implemented both synchronous and asynchronous algorithms
- Made use of a master-slave implementation
- System X from the Virginia Tech Terrascale Computing Facility
  - 1100 dual processor 2.3 GHz compute nodes
  - 4GB of RAM and 40 GB hard disk per node
- Used up to 32 processors
- Each run was repeated 10 times to account for variability in the results
Parallel Results: Accuracy

All results averaged over 10 runs
Parallel Results: Efficiency

\[ \text{Speedup} = \frac{\text{Time}_{1\text{Proc}}}{\text{Time}_{n\text{Proc}}} \]

\[ \text{Efficiency} = 100 \times \frac{\text{Speedup}}{n\text{Proc}} \]
Remarks

- PSO algorithm was able to produce consistent results in the presence of discrete design variables and severe numerical noise.
- PSO has significant potential for parallel processing:
  - Implementation is important.
  - Hardware is important.
- Asynchronous implementation provides almost perfect speedup:
  - 95% Efficiency with 32 processors.
  - Average time reduced from 150min (2.5 Hours) to 5 min.
Concluding Remarks

• Non-gradient based algorithms have a valuable place in design optimization
• These algorithms are often misused and should be applied with great care
• Rule of thumb is to only apply these algorithms if there is no alternative
• High computational cost can be alleviated by using parallel computing