การปรับปรุงขั้นตอนวิธีหาค่าเหมาะสมที่สุดแบบกลุ่ม โดยใช้ค่าที่ดีที่สุดในอดีต

Modified Particle Swarm Optimization with Historical Best Positions

ณัฐชัย สุธีรพงศ์พันธ์

ณัฐดนัย เปียสังข์

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรบัณฑิต

ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ปีการศึกษา 2559

การปรับปรุงขั้นตอนวิธีหาค่าเหมาะสมที่สุดแบบกลุ่ม โดยใช้ค่าที่ดีที่สุดในอดีต

Modified Particle Swarm Optimization with Historical Best Positions

ณัฐชัย สุธีรพงศ์พันธ์
ณัฐดนัย เปียสังข์

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตร
บัณฑิตภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
ปีการศึกษา 2559

ปริญญานิพนธ์ปีการศึกษา 2559

สาขาวิชาวิศวกรรมคอมพิวเตอร์

คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

เรื่อง การปรับปรุงขั้นตอนวิธีหาค่าเหมาะสมที่สุดแบบกลุ่มโดยใช้ค่าที่ดีที่สุดในอดีต

Modified Particle Swarm Optimization with Historical Best Positions

ผู้จัดทำ

1.นายณัฐชัย สุธีรพงศ์พันธ์ รหัสนักศึกษา 56010378

2. นายณัฐดนัย เปียสังข์ รหัสนักศึกษา 56010388

_____ อาจารย์ที่ปรึกษา

(รศ.ดร.บุญธีร์ เครือตราชู)

_____ อาจารย์ที่ปรึกษาร่วม

(รศ.กฤตวัน ศิริบูรณ์)

การปรับปรุงขั้นตอนวิธีหาค่าเหมาะสมที่สุดแบบกลุ่ม โดยใช้ค่าที่ดีที่สุดในอดีต

นายณัฐชัย	สุธีรพงศ์พันธ์	56010378
นายณัฐดนัย	เปียสังข์	56010388
รศ. ดร. บุญธีร์	เครือตราชู	อาจารย์ที่ปรึกษา
รศ. กฤตวัน	ศิริบูรณ์	อาจารย์ที่ปรึกษาร่วม

บทคัดย่อ

โครงงานนี้จัดทำขึ้นเพื่อศึกษา ออกแบบ และพัฒนาวิธีการหาค่าเหมาะสมที่สุด (Optimization Algorithm) ที่มีชื่อว่า การหาค่าเหมาะสมที่สุดแบบกลุ่มอนุภาค (Particle Swarm Optimization) ซึ่งการหาค่าเหมาะสมที่สุดแบบกลุ่มอนุภาคนี้ใช้ในการแก้ปัญหาทางโปรแกรมมิ่งที่ไม่เชิงเส้น (Nonlinear Programming Problems) เนื่องจากการหาค่าเหมาะสมที่สุดแบบกลุ่มอนุภาคดั้งเดิมมีจุดด้อยคือ ทำให้อนุภาคติดอยู่ในค่าเหมาะสมที่สุดเฉพาะที่ เพื่อแก้ปัญหานี้ เราได้ทำการพัฒนาการทำงานของการหาค่าเหมาะสมที่สุดแบบกลุ่มอนุภาค โดยการสร้างภาชนะของอนุภาคเพื่อสร้างคำตอบใหม่ในขณะที่อนุภาคอื่นๆกำลังค้นหาคำตอบ ซึ่งภาชนะนี้ ทำให้เราได้เหมาะสมที่สุดสากลของกลุ่มอนุภาคค่าใหม่ที่ดีกว่าเดิม

Modified Particle Swarm Optimization with Historical Best Positions

Mr. Nattachai	Suteerapongpan	56010378
Mr. Natdanai	Piasang	56010388
Assoc. Prof. Dr. Boontee	Kruatrachue	Advisor
Assoc. Prof. Kritawan	Siriboon	Co-Advisor

ABSTRACT

This paper presents modified Particle Swarm Optimization to avoid trapping in local optimum, premature convergence problem. Since all the particles have a tendency to move toward the same Global Best (Gbest), it is easily to trap in global local optimum (Global Best). To circumvent this problem, a pool of historical best solution is kept from positions generated by each particle. This solution pool is used to generate new solution using technique similar to Harmony Search. This solution is used to update the pool and also the Global Best of PSO. This pool is updated while the swarm is searching through the search space. The advantage provided by the pool is the diversity of generating new global best solution instead of following only a single Gbest from PSO. The experiment is tested over 20 benchmark functions. As the result of this experiment, MPSO-HB shows a significant superior performance over PSO.

กิตติกรรมประกาศ

ข้าพเจ้าตระหนักเป็นอย่างดีว่าปริญญานิพนธ์เล่มนี้เริ่มต้นได้ด้วยความช่วยเหลือจากบุคคลหลายท่าน ข้าพเจ้าจึงขออุทิศไว้ดังนี้ ขอขอบพระคุณ คุณแม่ และคุณพ่อของข้าพเจ้าที่ได้ให้การสนับสนุนการศึกษามาโดยตลอด ครูอาจารย์ ตลอดจนผู้ให้การสนับสนุนหรือมีส่วนร่วมในการสนับสนุนทุกท่าน โดยเฉพาะอย่างยิ่ง รศ.ดร.บุญธีร์ เครือตราชู ที่คอยให้คำปรึกษาและชี้แนะแนวทางมาโดยตลอดอย่างไม่ขาดตกบกพร่อง ข้าพเจ้าขอขอบพระคุณและขออภัยบุคคลท่านอื่นที่มีส่วนเคยให้ความช่วยเหลือแก่ข้าพเจ้าที่ข้าพเจ้าไม่ได้เอ่ยถึง และขอให้ท่านเหล่านั้นจงมีแต่ความสุขความเจริญยิ่งๆขึ้นไป

ณัฐชัย สุธีรพงศ์พันธ์

ณัฐดนัย เปียสังข์

สารบัญ

หน้า

สารบัญ (ต่อ)

สารบัญรูป

สารบัญรูป (ต่อ)

รูป หน้า

สารบัญตาราง

บทที่ 1
บทนำ

1.1 ความเป็นมา

วิธีการทางเมต้าฮิวริสติกนั้นเป็นขั้นตอนวิธีการที่ใช้ในการหาทางออกหรือคำตอบโดยใช้การสร้างค้นหา การสร้าง และการเลือกคำตอบใหม่ วิธีการทางเมต้าฮิวริสติกเหมาะแก่การใช้แก้ปัญหาเพิ่มประสิทธิภาพหรือหาค่าความเหมาะสม (optimization problem) ขั้นตอนวิธีหาค่าเหมาะสมที่สุดแบบกลุ่ม (particle swarm optimization) เป็นหนึ่งในวิธีการทางเมต้าฮิวริสติกที่ใช้หาคำตอบที่เหมาะสมโดยใช้อนุภาคจำนวนหนึ่งเรียกว่าฝูงอนุภาค (swarm) วิ่งค้นหาในพื้นที่บริเวณจำกัดเขตที่ต้องการหาคำตอบ ส่วนวิธีเชิงพันธุกรรมหรือ Genetic Algorithm (GA) เป็นหนึ่งในรูปแบบที่หลากหลายของขั้นตอนวิธีเชิงวิวัฒนาการ (Evolution Algorithm) และเป็นหนึ่งในอัลกอริทึมที่ถูกใช้มากหรือมากที่สุดในบรรดาวิธีเชิงวิวัฒนาการต่างๆ GA เป็นอัลกอริทึมหนึ่งที่ใช้แก้ปัญหา หรือค้นหาโดยมีอาศัยหลักการกลไกการการคัดเลือกโดยธรรมชาติของ ชาลส์ ดาร์วิน ผู้เป็นนักธรรมชาติวิทยาชาวอังกฤษ ในช่วงสองทศวรรศ ที่ผ่านมานั้นมีการรุดหน้าอย่างมากในกระบวนการเชิงตัวเลขและสถิติ รูปแบบกระบวนการ GA นั้นมีประโยชน์อย่างมากในการหาคำตอบที่เหมาะสมของปัญหาที่ไม่มีรูปแบบสมการที่ชัดเจนอย่างกว้างขวาง อีกทั้งยังถูกประยุกต์ใช้อย่างแพร่หลาย เช่น การประยุกต์ใช้ในการวิเคราะห์ความเสี่ยงต่อการล้มละลาย (Genetic algorithms applications in the analysis of insolvency) นอกจากนี้วิธีเชิงพันธุกรรมนั้นยังถูกนำมาพัฒนาในการความมุ่งหมายและวัตถุประสงค์ ทั้งนี้ขั้นตอนวิธีการหาค่าเหมาะสมที่สุดแบบกลุ่มมีข้อด้อยคือการลู่เข้าที่เร็วไป กล่าวคือฝูงอนุภาคเจอคำตอบเร็วเกินไปและอนุภาคในฝูงหยุดวิ่งค้นหาคำตอบต่อ ทำให้คำตอบที่ได้ยังเป็นคำตอบที่หยาบและเลวดังนั้นโครงงานวิจัยนี้จึงทำการศึกษาเพื่อปรับปรุงขั้นตอนวิธีการหาค่าเหมาะสมที่สุดแบบกลุ่ม

1.2 วัตถุประสงค์ของโครงงาน

1) เพื่อเสริมสร้างความรู้ ความเข้าใจในการหาคำตอบด้วยเมต้าฮิวริสติก (Metaheuristic)
2) เพื่อเรียนรู้และเข้าใจในขึ้นตอนวิธีเชิงพันธุกรรม (Genetic Algorithm)
3) เพื่อเรียนรู้และเข้าใจในการหาค่าที่ดีที่สุดแบบกลุ่ม (Particle Swarm Optimization)
4) สามารถนำความรู้ที่มีและที่ได้รับมาบูรณาการเพื่อต่อยอดและทำให้เกิดประโยชน์ได้
5) ทำการวิจัยเพื่อ ออกแบบ ปรับปรุง และพัฒนาวิธีการหาคำตอบด้วยเมต้าฮิวริสติก

6) เพื่อพัฒนาการดัดแปลงวิธีการคำตอบค่าเหมาะสมที่สุดของปัญหาต่างๆได้อย่างมีประสิทธิภาพทั้งในด้านความเร็ว และความแม่นยำในการหาคำตอบ

7) สามารถแก้ไขและลดปัญหาหรือลดผลกระทบของปัญหาในวิธีการที่ดัดแปลงขึ้นมาจากปัญหาที่มีอยู่ในวิธีการเดิมได้

1.3 สมมติฐานของการศึกษา

1) การผสมผสานระหว่างวิธีการของเมต้าฮิวริสติก 2 วิธี อันได้แก่ Genetic Algorithm และ Particle Swarm Optimization จะช่วยทำให้มีประสิทธิภาพมากขึ้น

2) คาดว่าวิธีนี้อาจจะเหมาะกับปัญหาบางประเภทเมื่อเทียบกับวิธีเดิม

3) ทดลองการนำการเคลื่อนที่แบบ PSO ใช้แทนหรือร่วมกับการไขว้เปลี่ยนทางพันธุกรรม (Cross over) จะเป็นการช่วยให้การหาค่าเหมาะสม แบบไม่เป็นเชิงเส้นได้หลากหลายมากขึ้น

1.4 วิธีดำเนินการ

1) ศึกษาความเป็นมาของวิธีเชิงพันธุกรรม

2) ศึกษา Particle swarm optimization

3) ศึกษาการพัฒนาการค้นหาคำตอบด้วย GA และ PSO ที่แล้วมา

4) ศึกษาวิธีการเปรียบเทียบให้เป็นมารตฐาน (benchmark)

5) ทำการทดลองปรับปรุง

6) วิเคราะห์การผลทดลอง

7) สรุปผลการทดลอง

8) เมื่อคำตอบที่ได้ไม่เป็นที่พอใจหรือยังไม่ดีพอ เริ่มกลับไปในขั้นตอนที่ 5)

9) เขียนผลการทดลองเมื่อผลการทดลองที่ได้เป็นที่พอใจแล้ว

1.5 ขอบเขตการวิจัย

ศึกษาเกี่ยวกับเมต้าฮิวริสติก (Metaheuristic) และวิธีการหาคำตอบต่างๆของเมต้าฮิวริสติกที่ได้รับแบบอย่างต่างกัน โดยจะเน้นที่ขั้นตอนวิธีเชิงพันธุกรรม (Genetic Algorithm) ทีได้รับแบบอย่างมาจากการคัดเลือกโดยธรรมชาติ (Natural Selection) การหาค่าที่ดีที่สุดแบบกลุ่ม (Particle Swarm Optimization) ที่ได้รับแบบอย่างมาจากการเคลื่อนที่ของฝูงนก

1.6 ประโยชน์ที่คาดว่าจะได้รับ

1) ได้รับความรู้เกี่ยวกับการหาคำตอบค่าเหมาะสมที่สุดโดยวิธีการของเมต้าฮิวริสติก

2) ได้รับประสบการณ์การศึกษาหาความรู้เพื่อการทำวิจัย

3) ได้นำความรู้ที่ได้รับไปใช้ในการหาค่าเหมาะสมที่สุดในงานวิจัยด้านอื่นๆ

บทที่ 2

ทฤษฎีพื้นฐานและงานวิจัยที่เกี่ยวข้อง

2.1 เมตาฮิวริสติก (Metaheuristic)

ในทางคณิตศาสตร์เราสามารถหาค่าความเหมาะสมที่สุด เช่นค่าต่ำสุด หรือค่าสูงสุดได้ด้วยการใช้วิธีการทางแคลคูลัสโดยการหาอนุพันธ์อันดับที่สอง แต่ในบางกรณีเราไม่สามารถใช้วิธีการนี้ได้ เช่น ปัญหามีความซับซ้อนสูง หรือปัญหาที่ไม่ทราบว่าสมการที่ใช้ในการหาคำตอบอยู่ในรูปแบบใด ในกรณีเช่นนี้วิธีการ Metaheuristic สามารถตอบโจทย์ได้ เมต้าฮิวริสติกเป็นวิธีการที่ใช้เพื่อหาคำตอบที่ดีที่สุด แต่อย่างไรก็ตามแม้ว่าวิธีการเมต้าฮิวริสติกจะสามารถหาคำตอบที่เหมาะสมและดีได้แต่ก็ไม่สามารถรับประกันความสำเร็จได้กล่าวคือไม่สามารถรับรองได้ว่าคำตอบที่ได้จะเป็นคำตอบที่ถูกต้องที่สุด

2.2 ขั้นตอนวิธีเชิงพันธุกรรม (Genetic Algorithm)

2.2.1 การหาค่าความเหมาะสมที่สุด

ขั้นตอนวิธีเชิงพันธุกรรม เป็นหนึ่งในวิธีการหาคำตอบของค่าที่เหมาะสมที่สุด โดยมีแนวคิดจากการคัดเลือกโดยธรรมชาติ (natural selection) แนวคิดการคัดเลือกโดยธรรมชาตินั้นเป็นแนวคิดที่ว่าด้วยประชากรแต่ละรุ่นจะถูกคัดสรรโดยธรรมชาติว่าจะให้อยู่ต่อจนเกิดการผสมพันธ์และได้ลูกหลานเกิดขึ้นมาใหม่ หรือตายไปขึ้ออยู่กับลักษณะความเหมาะสมของประชากรแต่ละตัว ประชากรที่มีลักษณะค่าความเหมาะสมมากก็จะมีโอกาสได้ผสมพันธ์มากและการผสมพันธ์จะทำให้การปรับเปลี่ยนทางพันธุกรรมที่แตกต่างไปโดยได้รับลักษณะทางพันธุกรรมจากพ่อและแม่ ซึ่งลูกทีเกิดมาอาจมีค่าความเหมาะสมที่ดีขึ้นหรือแย่ลงก็ได้ ซึ่งรุ่นลูกนี้ก็จะถูกคัดสรรโดยธรรมชาติให้มีอยู่รอดต่อหรือตายจากไป นอกจากการผสมพันธ์แล้วการกลายพันธ์จะทำให้เกิดความหลายหลายทางธรรมชาติรุ่นลูกที่เกิดการกลายพันธ์นี้จะมียีนที่แตกต่างไปจากเดิมโดยอาจเป็นยีนใหม่ที่ไม่เคยมีมาก่อนในประชากรรุ่นก่อนๆ

2.2.2 การแก้ปัญหาโดยใช้ขั้นตอนวิธีเชิงพันธุกรรม

การแก้ปัญหาโดยใช้ขั้นตอนวิธีเชิงพันธุกรรม หรือ Genetic Algorithm มักจะใช้กับปัญหาที่เป็น discrete โดยยีนประกอบด้วยโครโมโซมเป็นตัวควบคุมลักษณะทางพันธุกรรมซึ่งส่งผลต่อค่าความหมาะสมของประชากร (fitness) ซึ่งประเมินได้จากฟังก์ชันความเหมาะสม (fitness function) แต่ละตัวนั้นจะแทนด้วยค่าเลขฐานสอง (binary) ซึ่งมีค่า 0 และ 1

2.2.3 กระบวนการของ Genetic Algorithm และสิ่งที่เกี่ยว

2.3.3.1 การเข้ารหัส (Encoding)

เป็นการนำลักษณะคำตอบของปัญหาที่ต้องการหามาสร้างเป็นคำตอบที่อยู่ในรูปที่สามารถนำมาคำนวณได้ ซึ่งโดยมากนั้นคำตอบของขั้นตอนวิธีเชิงพันธุกรรมนั้นจะคำตอบอยู่ในรูปที่ไม่ต่อเนื่อง (discrete) เช่น ถ้าการหาค่าความเหมาะสมของสรีระนักว่ายน้ำ จะนำลักษณะทางร่างกาย ประสบการณ์ และปัจจัยอื่นๆของนักว่ายน้ำมาแสดงในรูปแบบที่นำมาประมวลผลได้ โดยที่หนึ่งรูปแบบของคุณลักษณะนำมาแทนด้วย 1 บิต

2.3.3.2 ยีน (Gene) และโครโมโซม (Chromosome)

เมื่อนำคำตอบของปัญหามาจำลองในรูปที่สามารถนำมาคิดหาคำตอบได้ สิ่งนั้นจะถูกเรียกว่ายีนซึ่งเมื่อยีนมีตั้งแต่ 1 ตัวขึ้นไปจะถูกเรียกว่าโครโมโซม โครโมโซมนี้เองเป็นตัวกำหนดลักษณะทางพันธุกรรมของประชากรหรือค่าเหมาะสมของประชากร

โครโมโซมรอบบอกของนักว่ายน้ำ 80 เซนติเมตร

1	0	1	0	0	0	0

รูป 2.1 โครโมโซมของคำตอบรอบบอกที่ส่งผลต่อสรีระที่เหมาะสมของนักว่ายน้ำ

2.3.3.3 การคัดสรร (selection)

การคัดสรรเป็นกระบวนการที่สำคัญเนื่องจากมีหน้าที่ในการคัดเลือกประชากรที่มีความเหมาะให้อยู่รอดเพื่อให้มีโอกาสได้ถ่ายทอดลักษณะทางพันธุกรรมที่ดีของตนเองไปยังลูกหลานและถ้ายีนที่ดีถูกถ่ายทอดมายังลูกหลาน ลูกหลานนั้นก็จะมีโอกาสได้อยู่รอดในรุ่นต่อๆสูง และเนื่องจากประชากรในแต่ละรุ่นจะมีจำนวนมากขึ้นกว่ารุ่นที่แล้วจึงจำเป็นต้องมีการคัดเลือกเพื่อให้ประชากรมีจำนวนคงที่ในรุ่นหนึ่งๆ การคัดเลือกดังกล่าวนี้ก็คือ selection

2.3.3.4 การผสมพันธ์ (recombination)

ประชากรในแต่ละรุ่นนั้นจะเกิดการสืบพันธ์เพื่อให้เกิดประชากรรุ่นลูกที่แตกต่างไปจากพ่อแม่โดยมีเป้าหมายเพื่อให้เกิดประชากรที่มีลักษณะดีขึ้นโดยประชากรรุ่นนั้นๆที่มีลักษณะค่าความเหมาะสมที่ดีก็จะมีโอกาสได้ผสมพันธ์สูง ส่วนประชากรที่มีลักษณะค่าความเหมาะสมที่ไม่

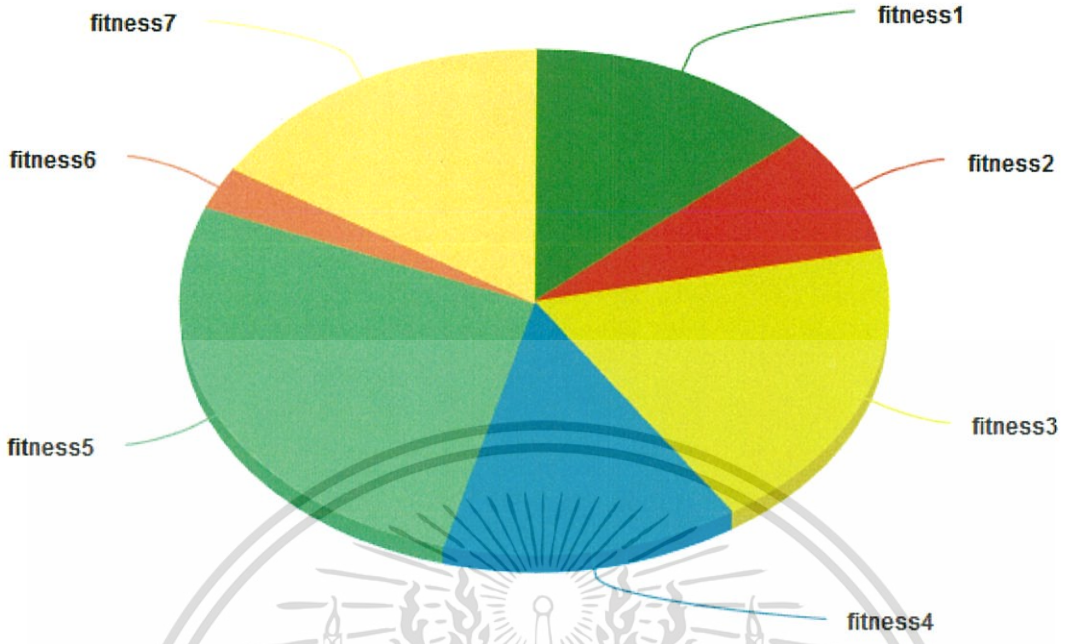
ดีก็ยังมีโอกาสได้ผสมพันธ์ได้ เนื่องจากมีแนวคิดที่ว่าพ่อแม่ที่มีลักษณะที่ดี ลูกที่เกิดจากการผสมพันธ์จะมีลักษณะที่ดี แต่ในบางครั้งพันธุกรรม (gene) ที่ดีอาจจะอยู่ในพ่อแม่ที่มีลักษณะที่ไม่ดี

2.3.3.5 การกลายพันธุ์ (mutation)

การกลายพันธุ์มีจุดประสงค์เพื่อดำรงค์ไว้ซึ่งความหลากหลายทางพันธุกรรมของประชากร องค์ประกอบของยีนของประชากรรุ่นใหม่ซึ่งเกิดจากการกลายพันธ์จะได้รับซึ่งอิทธิพลจากประชากรรุ่นก่อน เพราะเป็นการเปลี่ยนแปลงจากตัวเองและยีนที่เกิดการเปลี่ยนแปลงนั้นอาจจะไม่เคยมีมาก่อนในรุ่นก่อนหน้าทั้งหมด ซึ่งถ้าเป็นยีนที่ดีก็จะมีโอกาสสูงที่จะได้อยู่รอดต่อไปตลอดจนถ่ายทอดสู่รุ่นต่อไปได้ และในทำนองเดียวกันหากเป็นลักษณะทางพันธุกรรมที่ไม่ดีก็มีอาสสูงที่จะตายจากการคัดเลือกโดยธรรมชาติ การกลายพันธ์นั้นมีได้หลายรูปแบบขึ้นอยู่กับการนำขั้นตอนเชิงพันธุกรรมมาประยุกต์ใช้ เช่น การกลายพันธุ์ระดับบิต

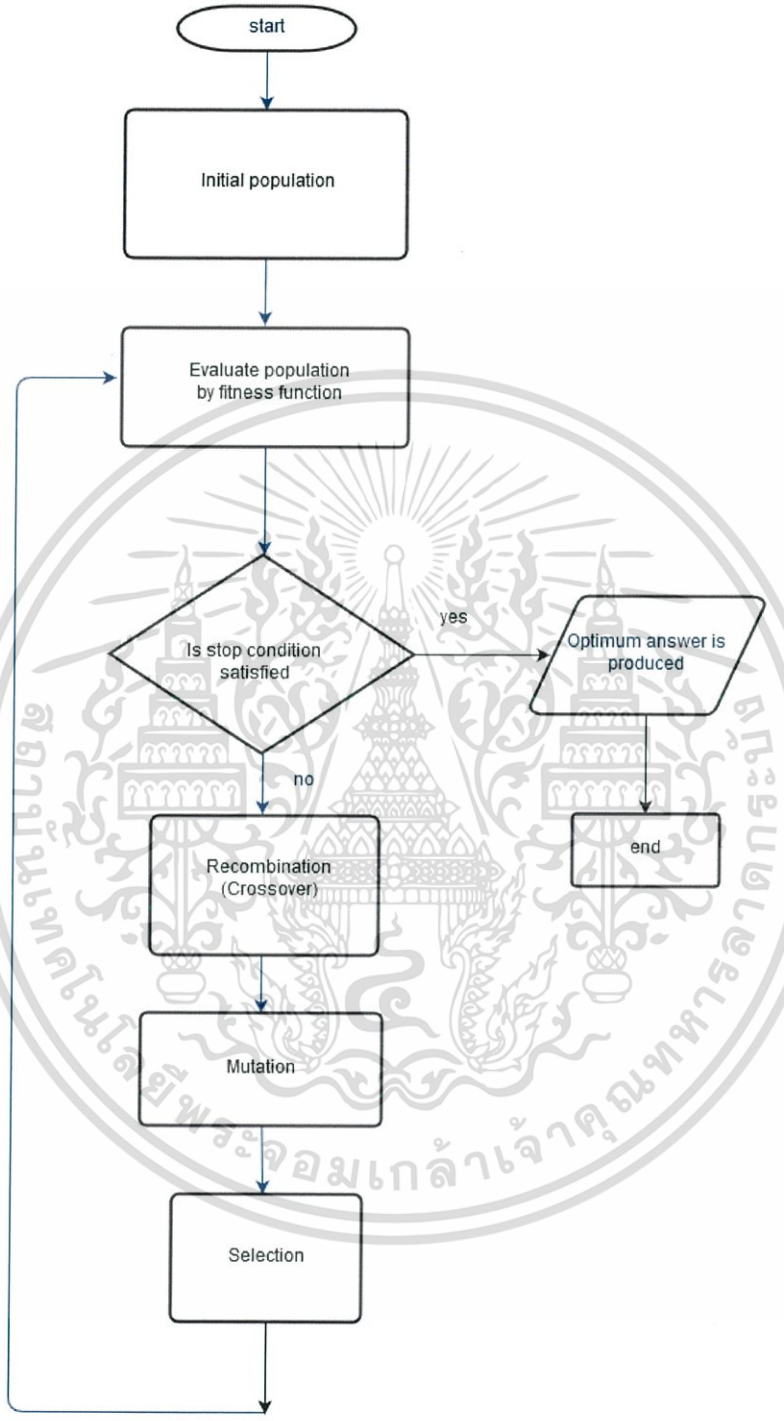
2.3.3.6 Roulette Wheel

ในการคัดเลือกผู้ที่จะมีสิทธิ์ได้ผสมพันธุ์นั้นหากเราให้ผู้ที่มีค่าความเหมาะสมที่ดีได้ผสมพันธุ์กันเองอย่างเดียว ยีนที่ดีแต่อยู่ในประชากรที่มีค่าความเหมาะสมก็จะถูกคัดทิ้งไปและจะไม่เป็นผลดีต่อการดำรงค์ไว้ซึ่งความหลากหลายทางธรรมชาติ จากแนวคิดที่ว่ายีนที่ดีอาจจะอยู่ในประชากรที่มีค่าความเหมาะสมไม่ดีนั้นการคัดสรรพ่อแม่ที่จะได้ผสมพันธ์กันนั้นจึงใช้ Roulette Wheel โดยที่จะแบ่งสัดส่วนความน่าจะเป็นในการถูกเลือกโดยสัดส่วนของค่าความเหมาะสม (fitness) จากรูปข้างล่างนี้ประชากรที่มีค่าความเหมาะสมเป็น fitness5 จะมีโอกาสที่จะได้ผสมพันธุ์มากที่สุด



รูป 2.2 การจำลอง Roulette Wheel

2.3.3.7 สรุปการทำงานขั้นตอนวิธีเชิงพันธุกรรม (flowchart)



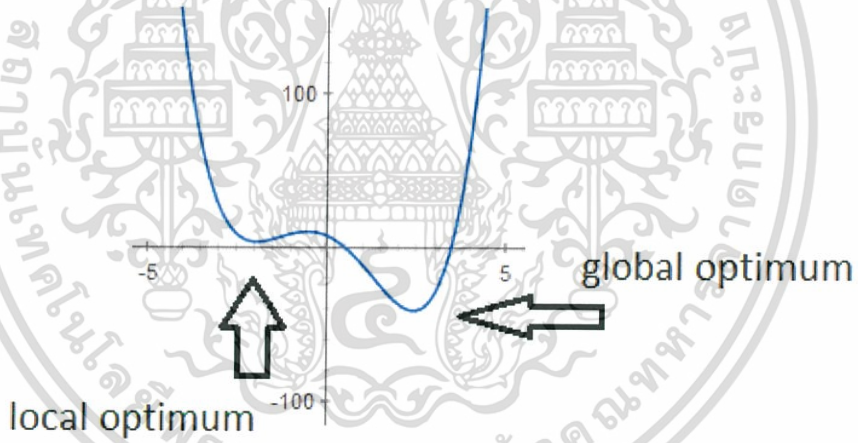
รูป 2.3 flowchart ของขั้นตอนวิธีเชิงพันธุกรรม

จากกระบวนการที่สำคัญดังที่ได้กล่าวมาจะสรุปเป็นขั้นตอนการทำงานได้ดังนี้

```
procedure genetic algorithm begin    t = 0
  initialize P(t)
  evaluate P(t)
  while (not termination-condition) do
begin
    t=t+1
    select P(t) from P(t-1)
    recombine P(t)
    evaluate P(t)
    end
end
```

รูป 2.4 ขั้นตอนการทำงานของขั้นตอนวิธีเชิงพันธุกรรม

2.3 ค่าความเหมาะสมสัมพัทธ์และสัมบูรณ์



รูป 2.5 ค่าเหมาะสมสัมพัทธ์และค่าเหมาะสมสัมบูรณ์

ค่าความเหมาะสมต่ำสุดสูงสุดมีอยู่ด้วยกันสองแบบคือ ค่าเหมาะสมอย่างสัมพัทธ์ และค่าเหมาะสมอย่างสัมบูรณ์ ในการหาค่าความเหมาะสมใน non-linear programming problem นั้น ปัญหาที่พบมากคือการติดกับดักที่ค่าความเหมาะสมอย่างสัมพัทธ์กล่าวคือ ในการหาคำตอบโดยใช้วิธีการหาค่าที่ดีที่สุดแบบกลุ่มนั้นมีโอกาสเป็นไปได้มากที่ฝูงนกเกิดการติดอยู่ที่ค่าเหมาะสมแบบสัมพัทธ์และไม่สามารถไปต่อได้จนเกิดการตายในที่สุด ในทำนองเดียวกันกับวิธีขั้นตอนเชิง

พันธุกรรมที่ใช้กับ non-linear programming problem เมื่อกลุ่มประชากรตกอยู่ในหลุมของ local optimum ทำให้มีโอกาสน้อยมากที่ประชากรตัวใดตัวหนึ่งจะเกิดการกลายพันธ์แล้วส่งผลให้หลุดออกมาจากหลุมนั้นและอยู่รอดต่อไปได้ในรุ่นถัดๆไป เพราะถึงแม้ว่าจะหลุดออกมาได้แต่ก็อาจยังไม่ถึงจุดที่มีค่าเหมาะสมมากกว่าประชากรตัวอื่นที่อยู่ในหลุมของค่าเหมาะสมแบบสัมพัทธ์ทำให้ถูกกระบวนการ selection คัดออกไป

2.4 การหาค่าที่ดีที่สุดแบบกลุ่ม (Particle Swarm Optimization)

PSO เป็นอีกหนึ่งในต้นแบบของ Evolutionary Computation ซึ่งมีพื้นฐานจากความฉลาดของฝูงนก PSO ถูกพัฒนาขึ้นโดยเจมส์ เคนเนดี้ (James Kennedy) และรัซเซล เอบเบอฮาท (Russell Eberhart) เขาได้นำเสนอแนวคิดสำหรับการออปติไมเซชัน (Optimization) ของฟังก์ชันที่ไม่เป็นเชิงเส้น (non-linear function) หลักการของ PSO นั้นดัดแปลงและลอกเลียนพฤติกรรมทางสังคมของสิ่งมีชีวิตทางธรรมชาติเช่น การออกหาอาหารของฝูงนก การว่ายของปลา พฤติกรรมดังกล่าวนั้นนำไปสู่ผลลัพธ์ที่ดีที่สุดของการแก้ปัญหาสมการที่ไม่เป็นเชิงเส้น

2.4.1 Particle Swarm Optimization

ในการนำแนวคิด PSO มาใช้นั้นโดยทั่วไปจะใช้การวิ่งของแต่ละอนุภาคในบริเวณของการค้นหา แต่ละอนุภาคนั้นจะมีการดึงดูดไปยังที่อนุภาคนั้นๆเคยผ่านมาและพบว่าดีที่สุด (personal best) และการดึงดูดอีกอย่างคือการดึงเข้าหาจุดๆที่ทุกๆอนุภาคเคยผ่านมาและพบว่าดีที่สุด (global best) ซึ่งอนุภาคที่เคยผ่านจุดที่ดีที่สุดในทุกๆจุดที่ทุกอนุภาคเคยผ่านจะถูกเรียกว่า gbest model ซึ่งทุกๆอนุภาคจะเชื่อมต่อเข้าหาและ ได้รับข้อมูลจากทุกๆอนุภาคในฝูง

การเคลื่อนที่ของนกแต่ละตัวนั้นจะขึ้นอยู่กับปัจจัยต่อไปนี้

1) ตำแหน่งของตัวมันเองในทุกๆมิติของบริเวณที่ต้องการค้นหาคำตอบ x_{id}

2) ตำแหน่งที่ดีที่สุดที่ซึ่งค้นพบโดยอนุภาคนั้นๆ p_{id}

3) ความเร็วของแต่ละตัว v_{id}

ในตอนแรกเริ่มแต่ละอนุภาคและความเร็วของตัวมันเองจะถูกกำหนดค่าโดยสุ่มให้อยู่ภายในบริเวณที่ต้องการค้นหา แต่ละอนุภาคจะวิ่งตลอดช่วงของบริเวณที่ต้องการค้นหาโดยมีการอัพเดทค่าความเร็วดังสมการที่ 1 และตำแหน่งของแต่ละอนุภาคจะถูกอัพเดทโดยสมการที่สอง การอัพเดทของทั้งสองสมการจะทำทุกมิติกับทุกๆอนุภาคตั้งแต่มิติที่ 1 ถึงมิติที่ n ดังแสดงรูปสมการดังต่อไปนี้

$$v_{id} = v_{id} + c \in_1 (p_{id} - x_{id}) + c \in_2 (p_{gd} - x_{id}) \quad (2.1)$$

$$x_{id} = x_{id} + v_{id} \quad (2.2)$$

เมื่อ x คือตำแหน่งของแต่ละอนุภาคในช่วงของการค้นหา

$$\vec{x}_i = (x_{i1}, x_{i2}, \dots, x_{id})$$

v คือค่าความเร็วของอนุภาคที่ i

$$\vec{v}_i = (v_{i1}, v_{i2}, \dots, v_{id})$$

p คือตำแหน่งที่ดีที่สุดแต่ละอนุภาคพบ (pbest)

$$\vec{p}_i = (p_{i1}, p_{i2}, \dots, p_{id})$$

$\in_1$ และ $\in_2$ เป็นค่าที่ถูกสุ่มขึ้นและเป็นอิสระต่อกัน

c คือค่าคงตัวที่มีค่าเท่ากับ 2.0

d คือมิติซึ่งมีค่าตั้งแต่ 1 ถึง n

PSO ในต้นแบบนี้จะมีปัญหาในด้านของการที่อนุภาคเกาะติดกับ vmax เนื่องจากค่าสุ่ม $\in_1$และ$\in_2$ นั้นจะทำให้ค่าความเร็วของอนุภาคเพิ่มขึ้นเรื่อยๆและส่งผลให้ตำแหน่งของอนุภาคนั้นเพิ่มขึ้นอย่างรวดเร็วจนลู่เข้าสู่อนันต์ในที่สุด

2.4.2 การลดอิทธิพลของความเร็วของอนุภาค

ต่อมาได้มีความพยายามในการปรับสมดุลระหว่างการออกสำรวจทั่วบริเวณทั้งหมด (global exploration) กับการตรวจสอบในบริเวณใกล้เคียง (local exploitation) พร้อมกับการหลีกเลี่ยงการเกาะติดกับความเร็วของอนุภาค (vmax) ความพยายามดังกล่าวได้นำมาซึ่งการเพิ่มพารามิเตอร์ความเฉื่อยเข้ามา หรือ Inertia Weight (ω) อย่างไรก็ตามการใช้ค่า vmax ค่าเดียวกับทุกปัญหานั้นเป็นการไม่เหมาะสม เช่น การใช้ค่า vmax น้อยกับการค้นหาในบริเวณที่กว้างจะทำให้การสำรวจนั้นไม่มากพอ และการใช้ค่า vmax มากกับการค้นหาในบริเวณที่น้อยจะส่งผลให้การสำรวจในบริเวณนั้นคล้ายกับการระเบิด เพราะฉะนั้นการเลือกค่า vmaxที่เหมาะสมนั้นเป็นสิ่งที่จำเป็น การเลือกค่าที่ไม่เหมาะสมจะส่งผลให้ประสิทธิภาพต่ำลงอย่างมาก และการเลือกดังกล่าวไม่ใช่เรื่องง่ายและอยู่นอกขอบเขตที่จะทำได้ด้วยการลองผิดลองถูก (trial and error) ด้วยการนี้

พารามิเตอร์ความเฉื่อยนั้นถูกเพิ่มขึ้นมาเพื่อแทนที่ vmax ด้วยการปรับอิทธิพลของความเร็วก่อนหน้าของอนุภาค จากการเพิ่มเข้ามาของค่าความเฉื่อยนี้ทำให้ได้สมการใหม่เป็น

$$v_{id} = \omega v_{id} + c \in_1 (p_{id} - x_{id}) + c \in_2 (p_{gd} - x_{id}) \qquad (2.3)$$

2.4.3 Constriction Factor ตัวประกอบ

อีกหนึ่งวิธีการที่จะกล่าวถึงที่ใช้ในการปรับสมดุลระหว่างการออกสำรวจทั่วบริเวณ กับการตรวจสอบในบริเวณใกล้เคียงนั้นคือการบีบรัด (constriction) ในปี 1999 มัวร์ริ แขรค์ (Maurice Clerc) ได้นำเสนอค่า constriction factor เพื่อเป็นการเพิ่มประสิทธิภาพการลู่เข้าของ PSO ในที่นี้จะนำเสนอเพียงรูปสมการหลังจากการเพิ่มค่าการบีบรัดเข้ามาและที่มาอย่างคร่าวๆส่วนรายละเอียดและที่มานั้นอยู่นอกเหนือขอบเขตของการศึกษา

$$v_{id} = K * [v_{id} + c_1 \in_1 (p_{id} - x_{id}) + c_2 \in_2 (p_{id} - x_{id})] \quad (2.4)$$

$$K = \frac{2}{2 - \varphi - \sqrt{\varphi^2 - 4\varphi}}, \quad \text{เมื่อ} \quad \varphi = c_1 + c_2, \varphi > 4 \qquad (2.5)$$

จากสมการที่ (2.5) นั้นพบว่าเมื่อใช้ค่า φ (ฟี) น้อยกว่า 4.0 จะเกิดการเคลื่อนที่วนรอบเข้าหาจุดที่ดีที่สุดในบริเวณของการค้นหา (search space) คล้ายกับรูปแบบก้นหอย (spiral) ซึ่งการเคลื่อนที่ดังกล่าวไม่สามารถรับรองการลู่เข้า (convergence) ได้ ในขณะที่เมื่อใช้ค่า $\varphi > 4$ จะส่งผลให้การลู่เข้านั้นรวดเร็วและถูกรับรอง จากสมการที่ (2.5) จะทำให้เราได้ค่า k เป็น 0.729 และ c_1, c_2 เป็น 1.49445

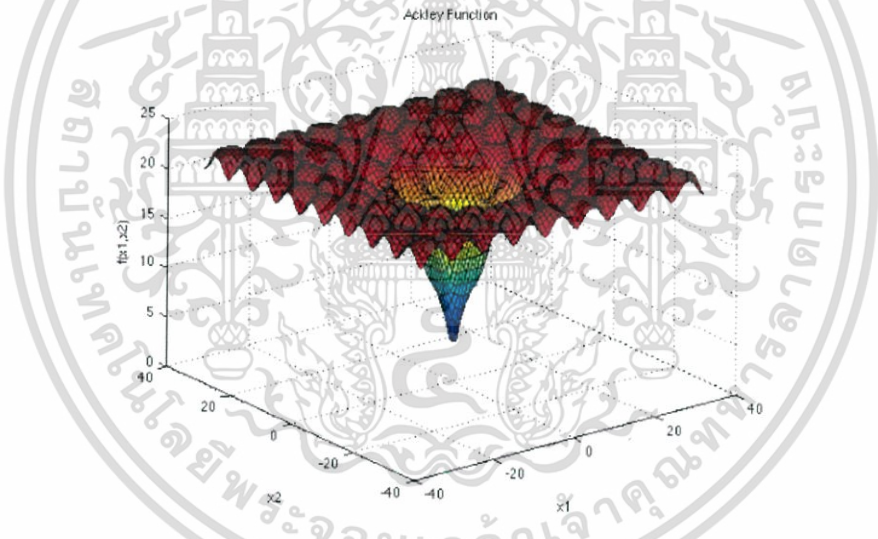
*หมายเหตุ ข้อความบางส่วนในหัวข้อ (2.4) หน้า 10-12 ได้ถอดความมาจากบทความตีพิมพ์ดังต่อไปนี้ (1) Bratton, Daniel, and James Kennedy. "Defining a standard for particle swarm optimization." 2007 IEEE swarm intelligence symposium. IEEE, 2007. (2) Shi, Yuhui. "Particle swarm optimization: developments, applications and resources." evolutionary computation, 2001. Proceedings of the 2001 Congress on. Vol. 1. IEEE, 2001.

บทที่ 3
การทดสอบประสิทธิภาพอัลกอริทึม

3.1 ฟังก์ชันที่ใช้ในการทดสอบ (benchmark function)

ในการทดสอบประสิทธิภาพอัลกอริทึมนั้นโดยทั่วไปจะทดสอบด้วย benchmark function ซึ่งฟังก์ชันนี้เป็นฟังก์ชันที่มีปรับเปลี่ยนจำนวนมิติได้ มีจำนวนจุดต่ำสุดเฉพาะที่(local optima)มาก และยิ่งถ้าเราใช้จำนวนมิติมากเท่าไรจำนวน local optima ก็จะมากขึ้น ด้วยเหตุนี้ทำให้ผลจากการทดสอบด้วย benchmark function สามารถบ่งบอกประสิทธิภาพของวิธีการเมต้าฮิวริสติกได้ ในบทนี้เราจะนำเสนอฟังก์ชันที่เป็นมาตรฐานที่มักถูกทดสอบโดยงานวิจัยต่างๆอยู่บ่อยครั้งซึ่งมีด้วยกัน 6 ฟังก์ชัน

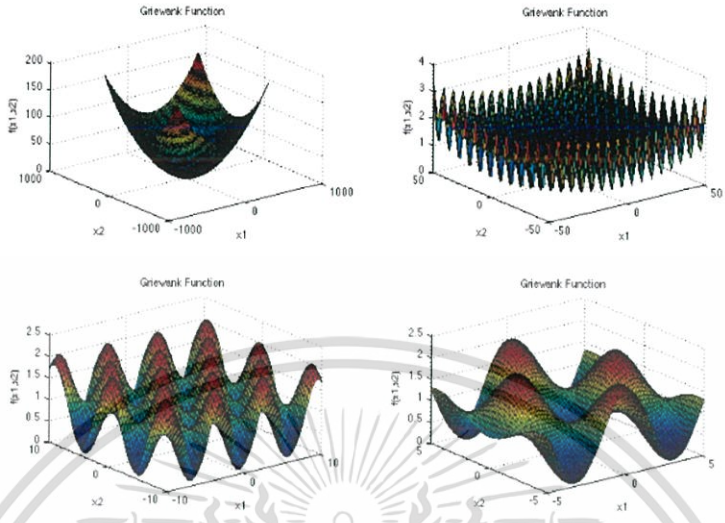
3.1.1 Ackley Function



$$f(\mathbf{x}) = -a\exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d}x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + \exp(1)$$

รูป 3.1 Ackley Function

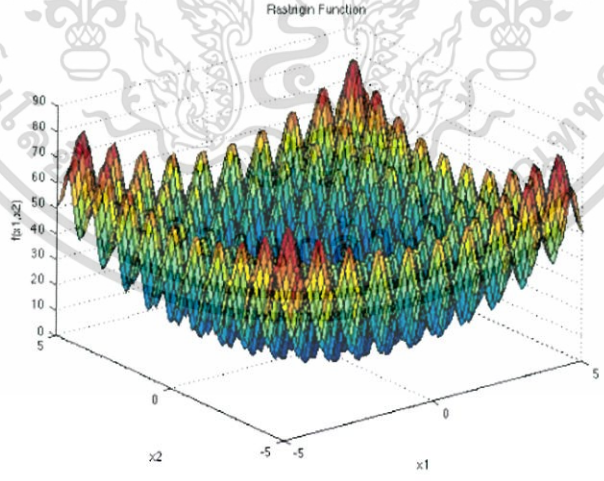
3.1.2 Griewank Function



$$f(\mathbf{x}) = \sum_{i=1}^{d} \frac{x_i^2}{4000} - \prod_{i=1}^{d} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

รูป 3.2 Griewank Function

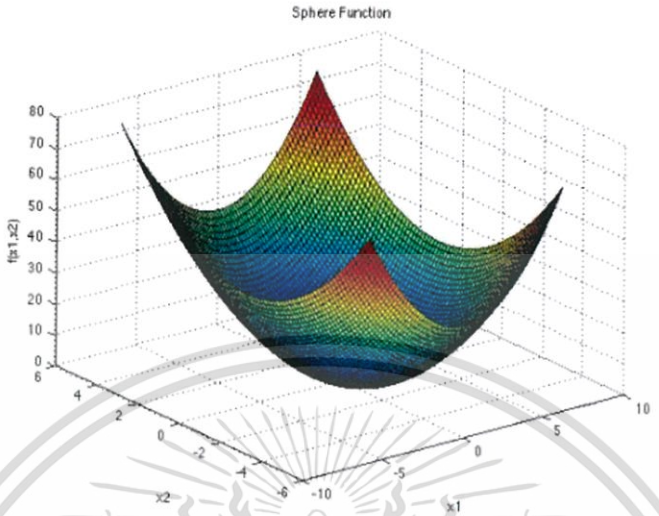
3.1.3 Rastrigin Function



$$f(\mathbf{x}) = 10d + \sum_{i=1}^{d} \left[x_i^2 - 10\cos(2\pi x_i)\right]$$

รูป 3.3 Rastrigin Function

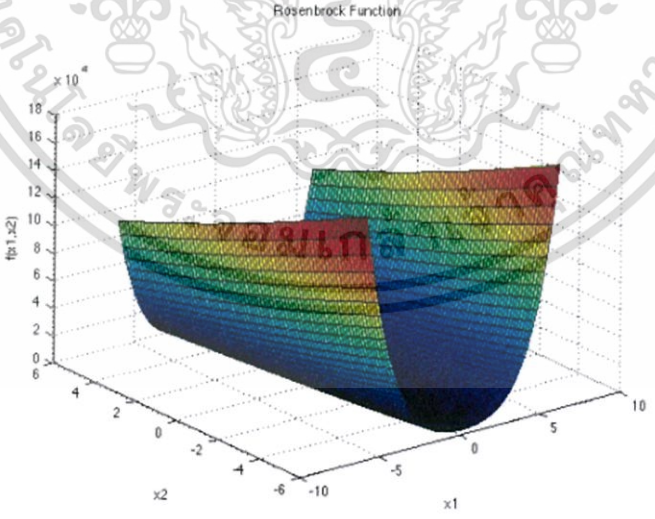
3.1.4 Sphere Function



$$f(\mathbf{x}) = \sum_{i=1}^{d} x_i^2$$

รูป 3.4 Sphere Function

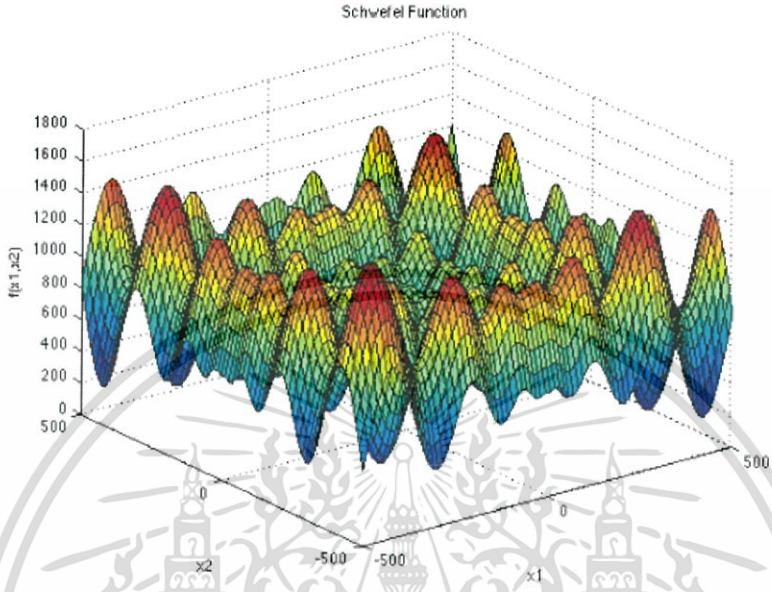
3.1.5 Rosenbrock Function – รูป 3.5



$$f(\mathbf{x}) = \sum_{i=1}^{d-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$$

รูป 3.5 Rosenbrock Function

3.1.6 Schwefel Function



$$f(\mathbf{x}) = 418.9829d - \sum_{i=1}^{d} x_i \sin(\sqrt{|x_i|})$$

รูป 3.3 Schwefel Function

3.2 การทดลอง

เราทำการศึกษาและทดลองปรับปรุง Particle Swarm Optimization โดยการเพิ่มกลุ่มประชากรเข้าไป เรียกว่ากลุ่มค่าที่ดีที่สุดในอดีต เพื่อที่ใช้ในการช่วยหาคำตอบของปัญหา โดยเทคนิคที่ใช้คือการสร้างคำตอบใหม่ โดยมีพันธุกรรมจากประชากรในกลุ่มค่าที่ดีที่สุดในอดีตนี้

บทที่ 4

PSO และกลุ่มค่าที่ดีที่สุดในอดีต

ในบทนี้จะพูดถึงขั้นตอนการทำงานของขั้นตอนวิธีหาค่าเหมาะสมที่สุดแบบกลุ่มอนุภาค (Particle Swarm Optimization : PSO) และอัลกอริทึมที่ปรับปรุงแล้วเรียกว่าขั้นตอนวิธีหาค่าเหมาะสมที่สุดแบบกลุ่มอนุภาคโดยใช้ค่าที่ดีที่สุดในอดีต (Particle Swarm Optimization with Historical Best positions : PSO-HB) การทำงานของ PSO นั้นมีข้อด้อยดังที่กล่าวไปคือการลู่เข้าที่เร็วเกินไป ทำให้ติดใน Local optimum ง่าย ในการปรับปรุง PSO จึงได้ทำการเพิ่มกลุ่มของคำตอบที่มีค่าความเหมาะสมดีที่สุดในอดีตจำนวนหนึ่ง คำตอบ นั้นจะเก็บแยกต่างหากใน pool ออกจากฝูงนกและถูกใช้ในการสร้างคำตอบใหม่ และถ้าคำตอบนี้ดีกว่าค่า GBest เดิม ก็จะมีการ set ค่า GBest ใหม่ ซึ่งจะมีผลกับการเคลื่อนที่ของ particle ของ PSO ในรอบถัดไป

การทำงานของ PSO จะเริ่มจากการกำหนดจำนวนประชากร (อนุภาค, particle) ขอบเขตของการค้นหา เงื่อนไขการหยุด เช่น ประชากร 50 ตัว ขอบเขตการค้นหา (-32, 32) และหยุดเมื่อ evaluate ครบ 300,000 ครั้ง จากนั้นจะกำหนดตำแหน่งของประชากรเริ่มต้นด้วยการสุ่มค่าในขอบเขตที่กำหนดไว้ เช่น (16, 32) และทำการ evaluate หาค่าความเหมาะสมของอนุภาค ซึ่งจะนำค่านั้นมาปรับปรุงค่าเหมาะสมที่สุดเฉพาะตัว (PBest) ของอนุภาคนั้น และค่าเหมาะสมที่สุดของกลุ่ม (GBest) จากนั้นก็จะทำกระบวนการเดิมที่อนุภาคถัดไปจนครบทุกอนุภาคในประชากร จากนั้นก็ทำการเคลื่อนย้ายอนุภาคแต่ละตัวในฝูงเข้าหา PBest และ GBest แล้วทำการ evaluate หาค่าความเหมาะสมของอนุภาคหลังการเคลื่อนที่และทำการอัพเดทค่า GBest และ PBest ของอนุภาค (กรณีที่อนุภาคหลุดออกนอกขอบเขตการค้นหาจะทำการข้ามขั้นตอน evaluation และไม่อัพเดท GBest และ PBest ของอนุภาคตัวนั้น) เมื่ออนุภาคทุกตัวเคลื่อนที่ครบจะนับเป็น 1 รอบการวนซ้ำ (iteration) จากนั้นจึงเริ่มการวนซ้ำรอบถัดไปจนกว่าจะถึงจุดที่ตรงกับเงื่อนไขการหยุดที่กำหนดไว้

การทำงานของ PSO-HB จะมีการสร้างคำตอบจาก 2 กระบวนการ โดยกระบวนการแรกสร้างจาก PSO กระบวนการที่สอง สร้างจากการสุ่มค่าในแต่ละ Dimension จากคำตอบใน pool ที่เก็บไว้ ทั้งสองกระบวนการนี้ มีผลกับการสร้างคำตอบใหม่ซึ่งกันและกัน โดยคำตอบที่ได้จากการเคลื่อนที่ของ PSO ถ้าได้คำตอบที่ดี จะนำมาแทนคำตอบที่แย่ใน pool ส่วนคำตอบที่ได้จากการสุ่มใน pool ก็จะนำมาแทนคำตอบที่แย่ใน pool เช่นกัน แต่กรณีที่ค่าที่ได้จากการสุ่มมีค่าดีกว่าค่า GBest ก็จะมีผลกับการสร้างคำตอบจากกระบวนการของ PSO ในรอบถัดไป การทำงานในแต่ละรอบก็จะ

สลับกันระหว่าง 2 กระบวนการ และจะทำงานในรอบถัดไปเรื่อยๆ จนกว่าจะถึงจุดที่ตรงกับเงื่อนไขการหยุดที่กำหนดไว้

การสุ่มคำตอบจากกลุ่มอดีตค่าที่ดีที่สุดนั้น น่าทำให้ได้คำตอบที่ดีขึ้น เนื่องจากค่าที่ดีใน dimension ใดๆ ของคำตอบที่ดีอยู่ในบริเวณใกล้เคียงกัน เราก็จะสุ่มค่าได้ในบริเวณนั้น เปรียบเสมือนเป็นการลงประชามติระหว่างคำตอบที่ดีแต่ละคำตอบว่าจะเชื่อคำตอบใด ถ้ามีคำตอบใน pool จำนวนมากอยู่ในบริเวณเดียวกัน ก็จะมีโอกาสสุ่มคำตอบได้ในบริเวณนั้นมากกว่าบริเวณอื่นที่มีคำตอบจำนวนน้อย และเมื่อทำการสุ่มทีละ dimension จนครบทุก dimension ก็จะได้คำตอบใหม่ โดยคำตอบนี้จะมีผลเช่นเดียวกับกับการผสมพันธุ์ของกระบวนการทางพันธุกรรม (Genetics Algorithm : GA) แต่จะมีพ่อแม่ คือ คำตอบที่ดีที่เก็บไว้ใน pool ทั้งหมด โดยคำตอบที่ได้ จะมีส่วนผสมมาจากหลายๆคำตอบใน pool โดยจะมีค่าในแต่ละ dimension เหมือนกับค่าใน dimension นั้นจากคำตอบส่วนใหญ่ใน pool

4.1 ขั้นตอนการทำงานของ PSO และ PSO-HB

4.1.1 การทำงานของ PSO

1) กำหนดประชากรเริ่มต้นด้วยจำนวนที่ต้องการ พร้อมทั้งประมวลผลหาค่าความเหมาะสมของแต่ละอนุภาค รวมทั้ง PBest ของอนุภาคแต่ละตัว และ GBest ของอนุภาคทั้งกลุ่ม
2) หาความเร็วของอนุภาคทุกตัวด้วยสมการ 2.1 ด้วย PBest และ GBest ที่หาได้
3) เคลื่อนที่อนุภาคทีละตัวจนครบทุกตัวด้วยสมการการเคลื่อนที่ 2.2 อัพเดท GBest และ PBest หลังจากที่อนุภาคแต่ละตัวเคลื่อนที่ และเมื่อครบรทุกตัวนับเป็น 1 รอบการวนซ้ำ
4) ทำซ้ำข้อ 3 จนกว่าจะเจอเงื่อนไขในการหยุด เช่น ค่า GBestไม่เปลี่ยนเป็นระยะเวลานานจนสามารถคาดการณ์ได้ว่าจะไม่เปลี่ยนอีก หรือ จำนวน iteration ครบตามกำหนด

4.1.2 การทำงานของ PSO-HB

1) กำหนดประชากรเริ่มต้นด้วยจำนวนที่ต้องการ พร้อมทั้งประมวลผลหาค่าความเหมาะสมของแต่ละอนุภาค รวมทั้ง PBest ของอนุภาคแต่ละตัว จากนั้น GBest ของอนุภาคทั้งกลุ่ม และเก็บค่า GBest ที่ได้ลงในกลุ่มอดีตค่าที่ดีที่สุด
2) หาความเร็วของอนุภาคทุกตัวด้วยสมการ 2.1 ด้วย PBest และ GBest ที่หาได้

3) เคลื่อนที่อนุภาคทีละตัวจนครบทุกตัวด้วยสมการการเคลื่อนที่ 2.2 อัพเดท PBest ของอนุภาคแต่ละตัวหลังจากที่อนุภาคเคลื่อนที่

- แต่ละครั้งที่เคลื่อนที่อนุภาค จะมีการเลือกว่าจะใส่ตำแหน่งของอนุภาคนั้นๆลงในกลุ่มอดีตค่าที่ดีที่สุดหรือไม่ โดยหากกลุ่มอดีตค่าที่ดีที่สุดยังมีสมาชิกไม่เต็มกลุ่มจะทำการใส่ลงในกลุ่มอดีตค่าที่ดีที่สุดเลย แต่หากสมาชิกกลุ่มเต็มแล้ว จะเปรียบเทียบค่าความเหมาะสมกับตำแหน่งที่แย่ที่สุดในกลุ่มอดีตค่าที่ดีที่สุด หากตำแหน่งที่แย่ที่สุดในกลุ่มอดีตค่าที่ดีที่สุดมีค่าความเหมาะสมดีกว่า จะไม่เก็บตำแหน่งใหม่ลงในกลุ่ม แต่หากค่าความเหมาะสมของตำแหน่งใหม่ดีกว่า จะทำการทิ้งคำตอบตำแหน่งที่แย่ที่สุดในกลุ่มอดีตค่าที่ดีที่สุด และใส่ค่าตำแหน่งใหม่ลงแทนที่ ตัวอย่างด้านล่างแสดงการเก็บคำตอบในกลุ่มค่าที่ดีที่สุดในอดีต

ลำดับ	fitness	x_1	x_2	x_3	…	x_d
0	1.686372	1.000897	0.998662	1.002304	…	0.099435
1	1.684983	1.000746	0.998637	1.002315	…	0.100247
2	1.683324	1.000799	0.998676	1.002297	…	0.100157
3	1.678852	1.000792	0.998729	1.002294	…	0.100717
4	1.665836	1.000408	0.998785	1.00137	…	0.100497
5	1.661147	1.000328	0.998853	1.001299	…	0.100673

ตาราง 4.1 กลุ่มค่าที่ดีที่สุดในอดีตก่อนเกิดการเปลี่ยนแปลง

ค่าความเหมาะสมของอนุภาคอนุภาคหนึ่งหลังการเคลื่อนที่คือ 1.67989 ซึ่งดีกว่าลำดับที่ 3 และแย่กว่าลำดับที่ 4 ดังนั้นค่าความเหมาะสมดังกล่าวจะแทนในลำดับที่ 3 จากนั้นคำตอบของค่าความเหมาะสม 1.678852 และคำตอบของค่าความเหมาะสมนั้นรวมถึงทุกค่าที่อยู่ลำดับมากกว่าคำตอบที่ถูกแทนที่จะถูกเลื่อนไปยังลำดับถัดไป

ลำดับ	fitness	x_1	x_2	x_3	...	x_d
0	1.686372	1.000897	0.998662	1.002304	...	0.099435
1	1.684983	1.000746	0.998637	1.002315	...	0.100247
2	1.683324	1.000799	0.998676	1.002297	...	0.100157
3	**1.67986**	**1.00089**	**0.99871**	**1.00226**	...	**0.100249**
4	1.678852	1.000792	0.998729	1.002294	...	0.100717
5	1.665836	1.000408	0.998785	1.00137	...	0.100497

ตาราง 4.2 กลุ่มค่าที่ดีที่สุดในอดีตหลังเกิดการเปลี่ยนแปลง

4) สร้างคำตอบใหม่โดยวิธีการสร้างคำตอบใหม่จะกล่าวโดยละเอียดต่อไป
5) ทำซ้ำข้อ 2 ถึงข้อ 4 จนกว่าจะเจอเงื่อนไขในการหยุด เช่น ค่า GBestไม่เปลี่ยนเป็นระยะเวลานานจนสามารถคาดการณ์ได้ว่าจะไม่เปลี่ยนอีก หรือ จำนวน iteration ครบตามกำหนด

4.2 กลุ่มอดีตค่าที่ดีที่สุด (Historical Best positions)

กลุ่มอดีตค่าที่ดีที่สุดถูกสร้างขึ้นมาเพื่อเก็บค่าที่เคยดีที่สุดในอดีต โดยจะมาจากการค้นพบของนกจากการเคลื่อนที่แต่ละครั้ง และอาจได้มาจากการสร้างคำตอบใหม่จากกลุ่มอดีตค่าที่ดีที่สุด ในการทดลองนี้กลุ่มอดีตค่าที่ดีที่สุดจะมีจำนวนอนุภาคทั้งหมด 50 อนุภาค ข้อมูลที่ถูกเก็บจะแยกต่างหากจากอนุภาคในฝูงนก โดยที่อนุภาคที่อยู่ในกลุ่มอดีตค่าที่ดีที่สุดจะมีเพียงข้อมูลของชุดคำตอบและค่าความเหมาะสมที่ได้จากการประเมินผล (evaluation) ของชุดคำตอบเท่านั้น ไม่มีการเก็บค่าความเร็ว และค่า PBest ของอนุภาคนั้นๆ และค่าที่ดีที่สุดในกลุ่มอดีตค่าที่ดีที่สุดก็จะหมายถึง GBest เพราะ GBest หมายถึงค่าที่ดีที่สุดเท่าที่หาเจอตลอดมา

4.3 การเก็บคำตอบของประชากรนกเข้าไปในกลุ่มอดีตค่าที่ดีที่สุด

การเก็บคำตอบเข้าไปในกลุ่มอดีตค่าที่ดีที่สุด จะมีการเปรียบเทียบระหว่างค่าความเหมาะสมของอนุภาคที่ต้องการจะเก็บกับค่าความเหมาะสมของอนุภาคที่อยู่ในกลุ่มอดีตค่าที่เหมาะสมที่สุดอยู่แล้ว หากตัวที่ต้องการจะเก็บดีกว่าตัวที่ถูกเปรียบเทียบก็จะมาแทนในตำแหน่งของตัวที่ถูกเปรียบเทียบ ตัวที่ถูกเปรียบเทียบและตัวถัดๆไปจะถูกเลื่อนไปหนึ่งลำดับส่งผลให้อนุภาคที่อยู่ในกลุ่มอดีตค่าที่ดีที่สุดตำแหน่งสุดท้ายล้นกลุ่มประชากรและก็จะถูกทิ้งไป แต่หากตัวที่

อยู่ในกลุ่มประชากรดีกว่าก็จะเทียบกับอนุภาคในกลุ่มอดีตค่าที่ดีที่สุดในลำดับถัดไปเรื่อยๆ จนกว่าจะเจอตำแหน่งที่สามารถแทนได้ หรือถ้าไม่สามารถแทนในตำแหน่งใดๆได้เลยก็จะถูกละทิ้งไป

นอกจากนี้ ยังได้มีการทดลองอัลกอริทึมที่สามารถตรวจสอบความแตกต่างของสมาชิกในกลุ่มอดีตค่าที่ดีที่สุดว่า ถ้าหากเหมือนกันมากเกินไป จะทำให้ค่าความหลากหลายภายในกลุ่มอดีตค่าที่ดีที่สุดลดลง และหาคำตอบได้ไม่ดีเท่าที่ควร วิธีคือ ใช้การตรวจค่าตำแหน่งอนุภาคในแต่ละมิติ หากค่าตำแหน่งในมิติหนึ่งๆของสองอนุภาคที่ตรวจสอบว่าห่างกันไม่เกินร้อยละ 10 ของหนึ่งในอนุภาค จะนับว่ามิตินั้นๆ เหมือนกัน และหากได้จำนวนมิติที่เหมือนกันมากกว่าร้อยละ 94 ของจำนวนมิติทั้งหมด จะนับว่าจุดสองจุดนั้นเป็นจุดเดียวกัน และใส่ค่าตำแหน่งที่ดีกว่าแทนค่าตำแหน่งที่แย่กว่าไปเลยโดยไม่เลื่อน Index ของอนุภาคตัวถัดๆไป

โปรแกรม 4.1.1 ฟังก์ชัน add_to_best_pool

```
bestPool[][]
FUNCTION add_to_best_pool(data)
     FOR j = 0 to bestPool.size -1
          IF dataFitness < BestPool[j].Fitness
               Shift data in bestPool by 1 time
               Set bestPool[j] to data
               Break from the FOR
          END IF
     END FOR
END FUNCTION
```

โปรแกรม 4.1.2 ฟังก์ชัน add_to_best_pool_with_similarity

```
bestPool[][]
FUNCTION add_to_best_pool_with_similarity(data)
     FOR j = 0 to bestPool.size -1
          IF dataFitness < bestPool[j].Fitness
               IF similar(data,j)
                    Set bestPool[j] to data
               ELSE
                    Shift data in bestPool by 1 time
                    Set bestPool[j] to data
               END IF
               Break from the FOR
          END IF
     END FOR
END FUNCTION
```

โปรแกรม 4.1.3 ฟังก์ชัน similar

```
bestPool[][]
FUNCTION similar(data,index)
     count = 0
     FOR j = 0 to bestPool.size -1
          IF ABS(data[j] - bestPool[index][j]) /
          bestPool[index][j]) < 0.1
               count = count + 1
          IF count > 0.94 * data.size
               RETURN TRUE
     END FOR
     RETURN FALSE
END FUNCTION
```

4.4 การสร้างคำตอบใหม่จากกลุ่มอดีตค่าที่ดีที่สุด

การสร้างคำตอบใหม่จากกลุ่มอดีตค่าที่ดีที่สุดจะใช้วิธีการสุ่มค่า Index i ของ pool และคำตอบใหม่ใน dimension นั้น จะมีค่าเท่ากับ ค่าใน dimension ของค่าใน pool ตำแหน่ง Index i ในแต่ละมิติ และนำค่าที่สุ่มได้ทุกค่ามาทำการหาค่า fitness และเก็บเข้าไปในกลุ่มอดีตค่าที่ดีที่สุด เช่นเดียวกับข้อ 4.3 หากคำตอบใหม่ดีกว่าค่าคำตอบใดในกลุ่มอดีตค่าที่ดีที่สุด ค่าคำตอบใหม่จะเข้าแทนที่ค่าคำตอบเก่า และขยับคำตอบเก่าลงไปใน Index ต่อๆไปตามลำดับ และค่าที่แย่ที่สุดในกลุ่มอดีตค่าที่ดีที่สุดจะถูกทิ้งไป ซึ่งในแต่ละรอบการเคลื่อนที่ (iteration, generation)แต่ครั้งจะมีการสุ่มคำตอบใหม่ 10 คำตอบ

โปรแกรม 4.2 ฟังก์ชัน generate_answer

```
bestPool[][]
FUNCTION generate_answer
     FOR i = 1 to 10
          data = random_generate
          dataFitness = evaluate(data)
          add_to_best_pool(data)
     END FOR
END FUNCTION
```

โปรแกรม 4.3 ฟังก์ชัน random_generate

```
FUNCTION random_generate
     FOR i = 0 to D
          r = Random  number between 0 to bestPool.size-1
          Set data[i] to bestPool[r][i]
     END FOR
END FUNCTION
```

4.5 อิทธิพลของกลุ่มอดีตค่าที่ดีที่สุด

กลุ่มอดีตค่าทีดีที่สุดถูกสร้างขึ้นมาให้เป็นอิสระจากกลุ่มประชากรอนุภาค (swarm) กล่าวคือ อนุภาคทุกตัวยังคงมีสมการที่ใช้ในการเคลื่อนที่อย่างปกติ แต่ค่าที่ดีที่สุด (GBest)ในขณะเวลาใดๆ อาจไม่ได้มาจากค่าที่ดีที่สุดของนกตัวใดๆในกลุ่มประชากร ค่าที่ดีที่สุดอาจเกิดได้จากการสร้างคำตอบใหม่จากกลุ่มอดีตค่าที่ดีที่สุดในรอบก่อนหน้า

4.6 ความหลากหลาย (diversity)

ความหลากหลาย (diversity) คือ ค่าความแตกต่างกันของประชากรแต่ละตัว โดยจะคิดจากผลรวมของความแตกต่างระหว่างค่าเฉลี่ยของตำแหน่งประชากรในแต่ละมิติและค่าตำแหน่งของประชากรแต่ละตัว หารด้วยจำนวนประชากร ซึ่งวิธีคิดนี้จะคล้ายกับการคิดส่วนเบี่ยงเบนมาตรฐานในทางสถิติ

ความหลากหลายเกี่ยวข้องกับอัลกอริทึมนี้คือ ความหลากหลายที่มีค่ามากขึ้นจะหมายถึงตัวอนุภาคจะกระจายตัวกันมากขึ้นและสามารถค้นหาคำตอบได้ดีขึ้น โดยสามารถคำนวณได้ดังนี้

$$diversity = \frac{1}{D}\sum_{j=1}^{D}\left(\frac{1}{S}\sum_{i=1}^{S}\left|x_i^j - \bar{x}^j\right|\right)$$

4.7 การจำกัดขอบเขตการค้นหา

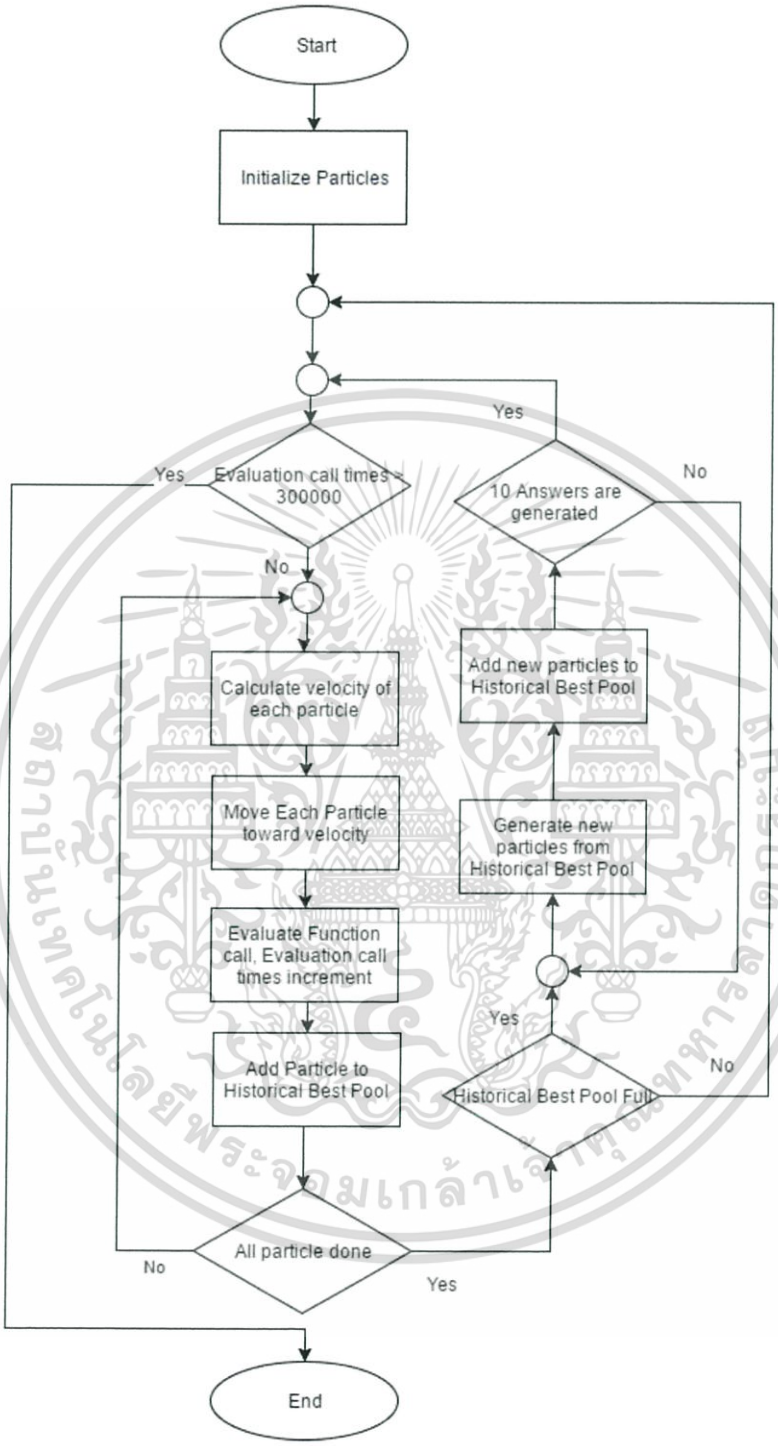
กรณีที่เราทราบถึงขอบเขตของคำตอบหรือคำตอบมีขอบเขตที่ตายตัว แต่ไม่กำหนดขอบเขตของการค้นหาคำตอบจะทำให้สิ้นเปลืองทรัพยากรในการค้นหาก็เหมือนกับการที่เราต้องการจะค้นหาของในโลกแต่กลับส่งยานอวกาศไปค้นหาคำตอบทั่วกาแลกซี เช่นนี้แล้วการกำหนดขอบเขตของการค้นหาจึงเป็นสิ่งที่จำเป็น เพราะฉะนั้นในการทดสอบอัลกอริทึมโดยใช้ benchmark function

จึงมักมีการกำหนดขอบเขตเอาไว้ นอกจากจะกำหนดขอบเขตของการค้นหาแล้วยังต้องมีการจัดการกับอนุภาคที่เคลื่อนที่ออกนอกขอบเขตด้วย

ในการเคลื่อนที่ของฝูงอนุภาคมีโอกาสที่อนุภาคจะเคลื่อนที่เกินขอบเขตการค้นหาอันเนื่องมาจากโมเมนตัมอันเกิดจากความเร็วครั้งก่อนทำให้อนุภาคมีความเร็วมากขึ้นเรื่อยๆ ในงานวิจัยหนึ่งได้นำเสนอให้กำหนดค่าความเร็วสูงสุด(v_{max}) ให้เท่ากับการใช้ค่า x_{max} จะทำให้การค้นหามีประสิทธิภาพ แต่อย่างไรก็ตามโอกาสที่อนุภาคจะเคลื่อนที่ออกนอกขอบเขตการค้นหาก็ยังเป็นไปได้อยู่ดี การกำหนดขอบเขตของการค้นหาเป็นสิ่งที่จำเป็น เช่นเดียวกันกับการจัดการกับอนุภาคตัวใดๆที่ซึ่งหลุดออกนอกขอบเขตการค้นหาด้วย ในการทดลองนี้เราได้ใช้วิธีการจัดการอย่างง่าย เมื่ออนุภาคตัวใดๆเคลื่อนที่ออกนอกขอบเขตการค้นหา การประเมินผลค่าความเหมาะสมของนกตัวนั้นจะถูกละเว้นส่งผลให้ค่าที่ดีที่สุดเฉพาะตัว (PBest) ไม่เปลี่ยนแปลง และในการบินครั้งถัดๆไปนกตัวดังกล่าวจะค่อยๆถูกดึงกลับมาและในที่สุดนกที่หลุดออกนอกขอบเขตการค้นหาก็จะกลับมาอยู่ในขอบเขตของการค้นหาอีกครั้ง

4.8 สรุปภาพรวมอัลกอริทึม

อัลกอริทึม PSO-HB สามารถสรุปเป็นภาพรวมได้ ดัง flow chart ต่อไปนี้



รูป 4.1 แผนภาพการทำงานของ PSO-HB

บทที่ 5

ผลการทดลอง

5.1 ปัญหาที่นำมาใช้ทดสอบ

ปัญหาที่นำมาใช้ทดลองอัลกอริทึมที่คิดค้นนั้น มีอยู่ทั้งหมด 6 ฟังก์ชั่น ดังที่ได้ที่กล่าวไปในข้างต้นในหัวข้อที่ 3.1 หน้า 13

5.2 ฟังก์ชั่นประเมินผล (Evaluate Function) และการควบคุม

ฟังก์ชั่นประเมินผล คือ ฟังก์ชั่นที่ใช้ในการหาค่า fitness เพื่อค้นหาค่าที่เหมาะสมที่สุดในฟังก์ชั่นนั้นๆ โดยฟังก์ชั่นที่นำมาใช้ในการทำ benchmark มีทั้งรูปแบบ Unimodal เช่น Sphere และ Multimodal ที่เป็นมาตรฐาน เช่น Ackley, Griewank, Rosenbrock และ Rastrigin และอีกหนึ่งที่เป็นที่นิยมคือ Schwefel รายละเอียดแสดงดังตาราง

ตาราง 5.1 ชื่อฟังก์ชัน จำนวนมิติและการกำหนดขอบเขตของการค้นหา

Equation	Name	D
$f_1(x) = -20exp\left\{-0.2\sqrt{\frac{1}{D}x_i^2}\right\} - exp\left\{\frac{1}{D}\sum_{i=1}^{D}\cos(2\pi x_i)\right\} + 20 + e$	Ackley	30
$f_2(x) = \frac{1}{4000}\sum_{i=1}^{D} x^2 - \prod_{i=1}^{D}\cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$	Griewank	30
$f_3(x) = \sum_{i=1}^{D}\{x_i^2 - 10\cos(2\pi x_i) + 10D\}$	Rastrigin	30
$f_4(x) = \sum_{i=1}^{D}(x_i)^2$	Sphere	30
$f_5(x) = \sum_{i=1}^{D-1}\{100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\}$	Rosenbrock	30
$f_6(x) = 418.9829D - \sum_{i=1}^{D}(x_i\sin(\sqrt{\lvert x_i\rvert})$	Schwefel	30

5.3 การกำหนดพื้นที่ที่ใช้ในการสุ่มประชากรเริ่มต้น

การกำหนดพื้นที่ของการสุ่มประชากรเริ่มต้นให้มีศูนย์กลางตรงกับค่าที่ดีที่สุด(global optimum)นั้นสามารถส่งผลอย่างเป็นนัยยะให้บางอัลกอริทึมมีประสิทธิภาพเกินควร และอาจจะเกิดความไม่เท่าเทียมในการเปรียบเทียบอัลกอริทึมที่ต่างชนิดกันได้ ซึ่งนับเป็นปัญหาหนึ่งที่สำคัญถ้าเราต้องการเปรียบเทียบอัลกอริทึม ในการรับแก้ปัญหาดังกล่าวมีหลายวิธีเช่น การปรับเปลี่ยนฟังก์ชันโดยเลื่อนจุดของglobal optimum เปลี่ยนไปจากเดิม เรียกว่า offset method อีกหนึ่งวิธีคือการปรับเปลี่ยนพื้นที่ของกลุ่มประชากรเริ่มต้นซึ่งเป็นวิธีที่จะใช้ในการทดลองนี้ วิธีดังกล่าวทำได้โดยการกำหนดพื้นที่ของกลุ่มประชากรเริ่มต้นให้เป็นพื้นที่ย่อย(sub-space) จากขอบเขตการค้นหา (feasible bound) และตำแหน่งของ global optimum ต้องไม่อยู่ในพื้นที่ย่อยดังกล่าว การทำเช่นนี้เรียกว่า region scaling

ตาราง 5.2 ฟังก์ชันและการกำหนดประชากรเริ่มต้น

ฟังก์ชัน	คำตอบที่ดีที่สุด	ค่าเหมาะสมที่สุด	ขอบเขตการค้นหาFeasible Bound	ขอบเขตประชากรเริ่มต้น
f_1	0.0^D	0.0	(-32, 32)	(16, 32)
f_2	0.0^D	0.0	(-600, 600)	(300, 600)
f_3	0.0^D	0.0	(-5.12, 5.12)	(2.56, 5.12)
f_4	0.0^D	0.0	(-300, 300)	(150, 300)
f_5	1.0^D	0.0	(-30, 30)	(15, 30)
f_6	420.9867^D	0.0	(-500, 500)	(-500, -250)

5.4 ตัวแปร และค่าคงตัวที่ใช้ในการทดลอง

ตัวแปรที่ใช้ในการทดลอง ได้มีการควบคุมไว้ดังนี้

ตัวแปรควบคุมสำหรับ Particle Swarm Optimization ธรรมดา

ตาราง 5.3 แสดงการควบคุมตัวแปรสำหรับ Particle Swarm Optimization

จำนวนอนุภาค	50
จำนวนครั้งในการทดลอง	100 ครั้งต่อฟังก์ชั่น
ค่าสัมประสิทธิ์ความเร็วต้น (ω)	0.729
ค่าสัมประสิทธิ์ความเร็วเข้าหา PBest (c_2)	1.496180
ค่าสัมประสิทธิ์ความเร็วเข้าหา GBest (c_1)	1.496180

ตัวแปรควบคุมสำหรับ Particle Swarm Optimization แบบมีกลุ่มประชากรอนุภาค

ตาราง 5.4 การควบคุมตัวแปรสำหรับ Particle Swarm Optimization with Historical Best

จำนวนอนุภาค	50
จำนวนอนุภาคในกลุ่มประชากร	50
จำนวนครั้งที่สุ่มหาคำตอบใหม่ต่อรอบ	10 ครั้ง
จำนวนครั้งในการทดลอง	30 ครั้งต่อฟังก์ชั่น
ค่าสัมประสิทธิ์ความเร็วต้น (ω)	0.729
ค่าสัมประสิทธิ์ความเร็วเข้าหา PBest (c_2)	1.496180
ค่าสัมประสิทธิ์ความเร็วเข้าหา GBest (c_1)	1.496180

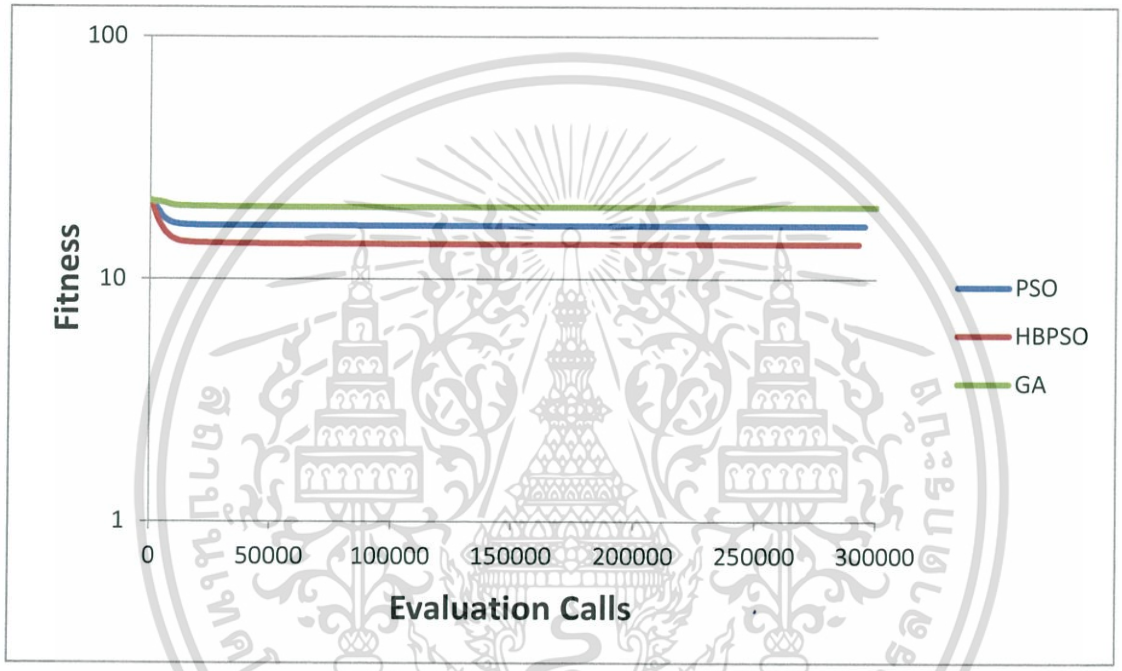
5.5 เปรียบเทียบผลการทดลองด้วยฟังก์ชันพื้นฐาน

การเปรียบเทียบประสิทธิภาพนั้นวัดโดยค่าความผิดพลาดที่น้อยที่สุดของค่าความเหมาะสม

$$|f(x) - f(x^*)|$$

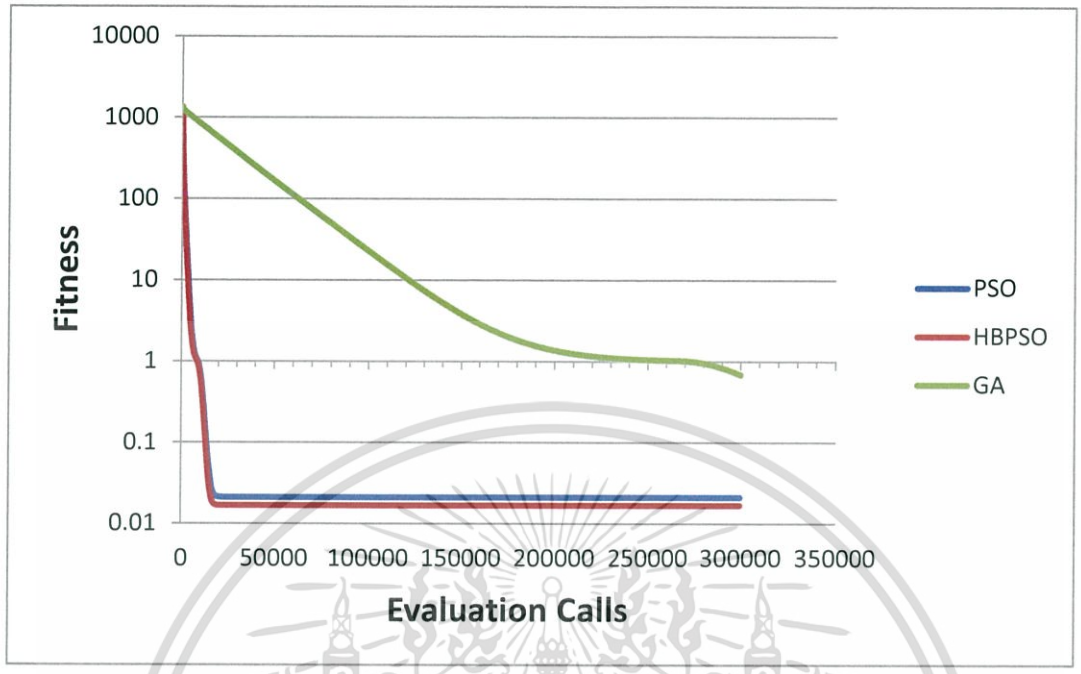
โดยที่ $f(x^*)$ คือค่าที่เหมาะสมที่สุด(optimum)ของปัญหานั้นๆ

5.5.1 Ackley Function



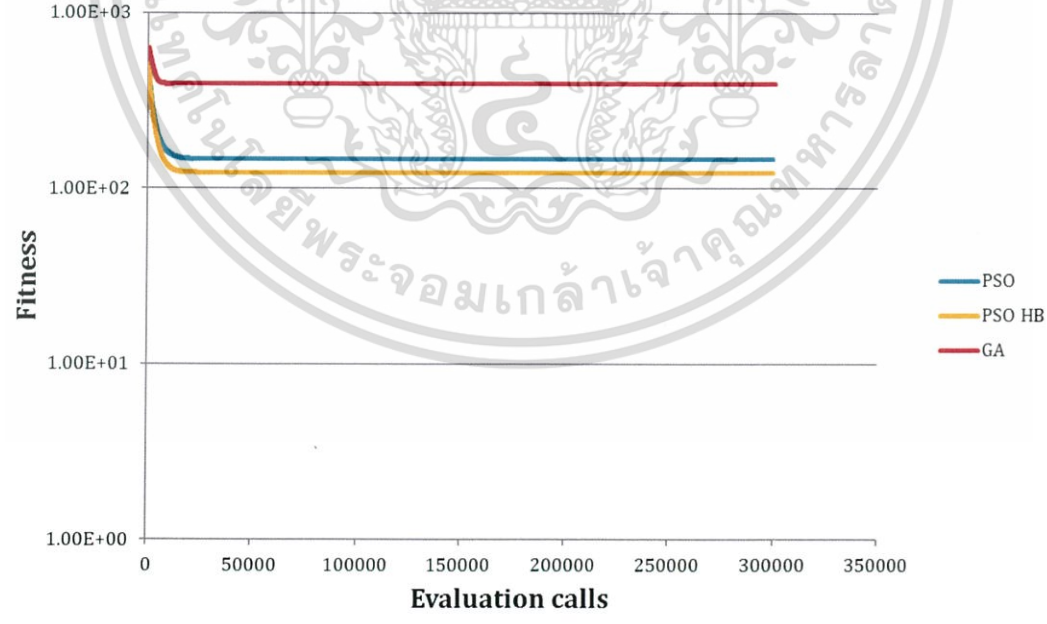
รูป 5.1 กราฟค่าความเหมาะสมในฟังก์ชัน Ackley

5.5.2 Griewank Function



รูป 5.2 กราฟค่าความเหมาะสมในฟังก์ชัน Griewank

5.5.3 Rastrigin Function

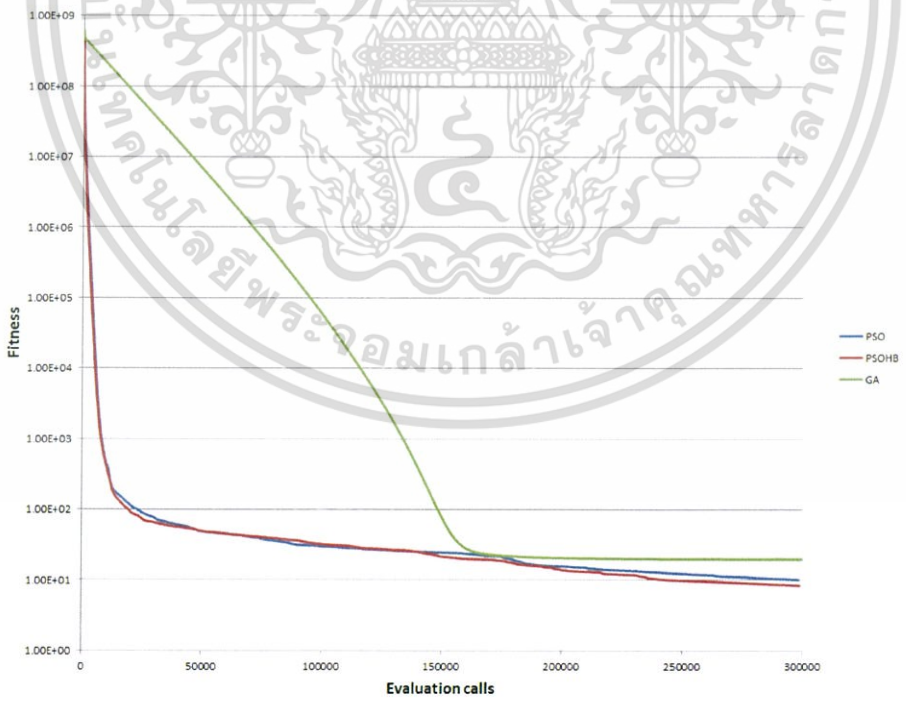


รูป 5.3 กราฟค่าความเหมาะสมในฟังก์ชัน Rastrigin

5.5.4 Sphere Function

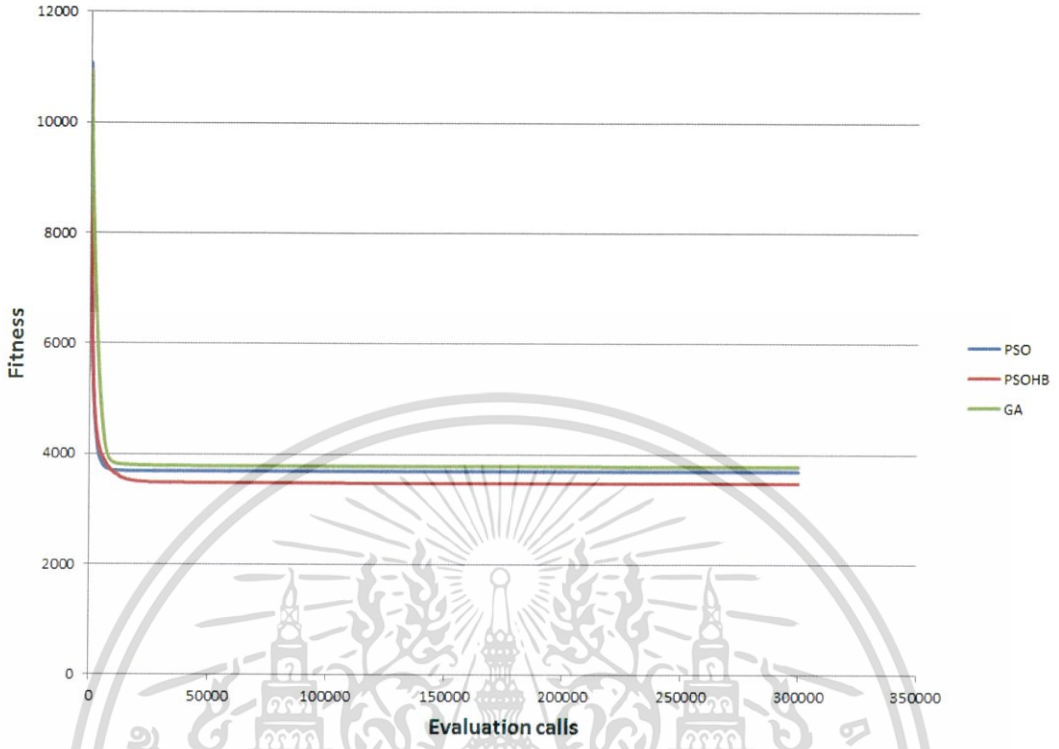
รูป 5.4 กราฟค่าความเหมาะสมในฟังก์ชัน Sphere

5.5.5 Rosenbrock Function



รูป 5.5 กราฟค่าความเหมาะสมในฟังก์ชัน Rosenbrock

5.5.6 Schwefel Function



รูป 5.6 กราฟค่าความเหมาะสมในฟังก์ชัน Schwefel

ในฟังก์ชันที่เป็นชนิด Multimodal เช่น Ackley และ Griewank จากผลการทดลองบ่งบอกว่า PSO มีโอกาสเกิดการติด Local optimum ที่มากกว่า และจากค่าความผิดพลาดของค่าความเหมาะสมบ่งชี้ว่าจำนวนจำนวนคำตอบที่ไม่ optimum มีจำนวนมากกว่า

5.6 ฟังก์ชันเพิ่มเติมที่ใช้ในการทดสอบ

นอกจากฟังก์ชันพื้นฐานที่เป็นที่นิยมใช้กันอย่างแพร่หลายแล้ว การทดลองนี้ยังได้ใช้เบนช์มาร์คฟังก์ชันเพิ่มเติมจาก 6 ฟังก์ชันพื้นฐานอีกเป็นจำนวน 9 ฟังก์ชัน ได้แก่ f_7 - f_{15} รายละเอียดของฟังก์ชัน ได้แก่ รูปสมการ คำตอบที่ดีที่สุด จำนวนมิติ จะแสดงในตารางต่อไปนี้

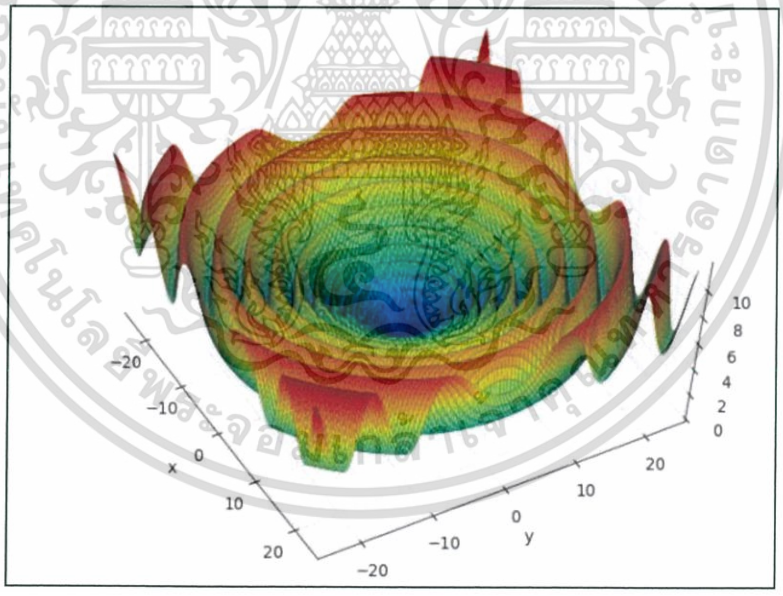
ตารางที่ 5.5 ฟังก์ชันเพิ่มเติมที่ใช้ทดสอบ

Equation	Name	D
$f_7(x) = 0.5 + \frac{sin^2(x^2+y^2 - 0.5)}{(1.0 + 0.001 * (x^2+y^2))^2}$	Schaffer	2
$f_8(x) = \sum_{i=1}^{D-1} {x_i}^2 + (\sum_{i=1}^{D} \frac{i}{2} x_i)^2 + (\sum_{i=1}^{D} \frac{i}{2} x_i)^4$	Zakharov	30
$f_9(x) = (1.5 - x_1 + x_1x_2)^2 + (2.25 - x_1 + x_1x_2^2)^2 + (2.65 - x_1 + x_1x_2^3)^2$	Beale	2
$f_{10}(x) = \sum_{i=0}^{D-1} [({x_i}^2)^{({x_{i+1}}^2+1)} + ({x_{i+1}}^2)^{({x_i}^2+1)}]$	Brown	30
$f_{11}(x) = {x_0}^2 + 10^6 \sum_{i=1}^{D} {x_i}^2$	Cigar	30
$f_{12}(x) = [1 + (x_1 + x_2 + 1)^2(19 - 14x_1 + 3{x_1}^2 - 14x_2 + 6x_1x_2 + 3{x_2}^2)] * [30 + (2x_1 - 3x_2)^2(18 - 32x_1 + 12{x_2}^2 + 48x_2 - 36x_1x_2 + 27{x_2}^2)]$	Goldstein-Piece	2
$f_{13}(x) = -\exp\left(-0.5 \sum_{i=1}^{n} x_i^2\right) + 1$	Exponential	30
$f_{14}(x) = \sum_{i=1}^{D} (i * {x_1}^2)$	Parallel Ellipsoid	30
$f_{15}(x) = \sum_{i=1}^{D} \sum_{j=1}^{i} ({x_i}^2)$	Rotated Ellipsoid	30

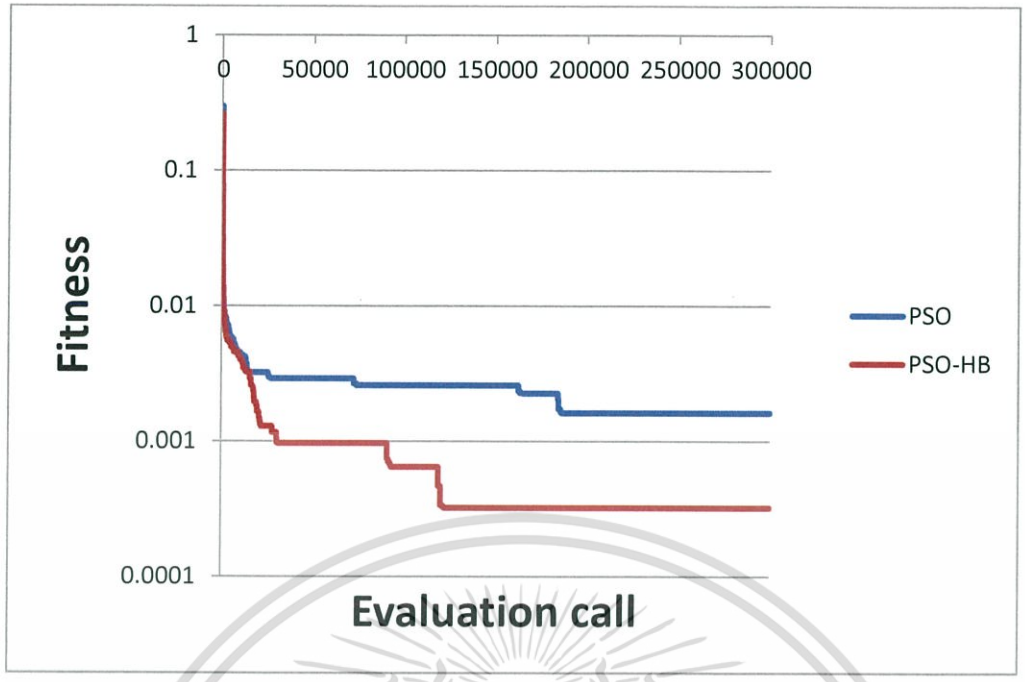
ตารางที่ 5.6 ฟังก์ชันและการกำหนดประชากรเริ่มต้น

ฟังก์ชัน	คำตอบที่ดีที่สุด	ค่าเหมาะสมที่สุด	ขอบเขตการค้นหา Feasible Bound	ขอบเขตประชากรเริ่มต้น
f_7	0.0^D	0.0	(-100, 100)	(-100, 100)
f_8	0.0^D	0.0	(-5, 10)	(-5, 10)
f_9	$2, 3.5$	0.0	(-4.5, 4.5)	(-4.5, 4.5)
f_{10}	0.0^D	0.0	(-1, 4)	(-1, 4)
f_{11}	1.0^D	0.0	(-10, 10)	(-10, 10)
f_{12}	$0, 1$	0.0	(-2, 2)	(-2, 2)
f_{13}	0.0^D	0.0	(-1, 1)	(-1, 1)
f_{14}	0.0^D	0.0	(-5.12, 5.12)	(-5.12, 5.12)
f_{15}	0.0^D	0.0	(-10, 10)	(-10, 10)

5.6.1 Schaffer Function

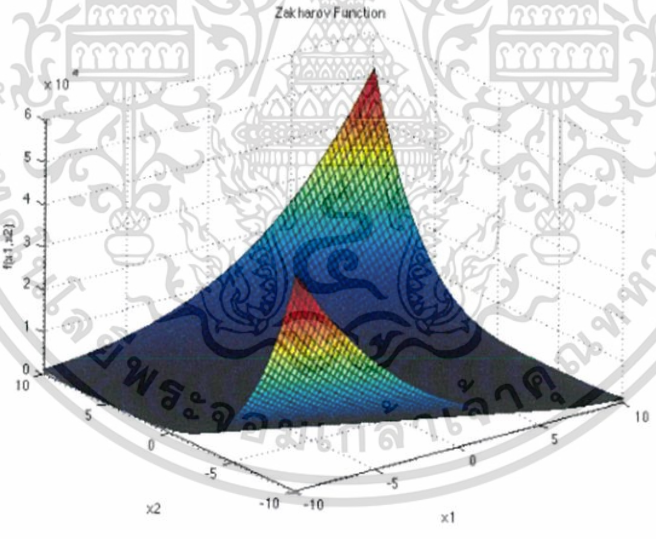


รูป 5.7 Schaffer Function



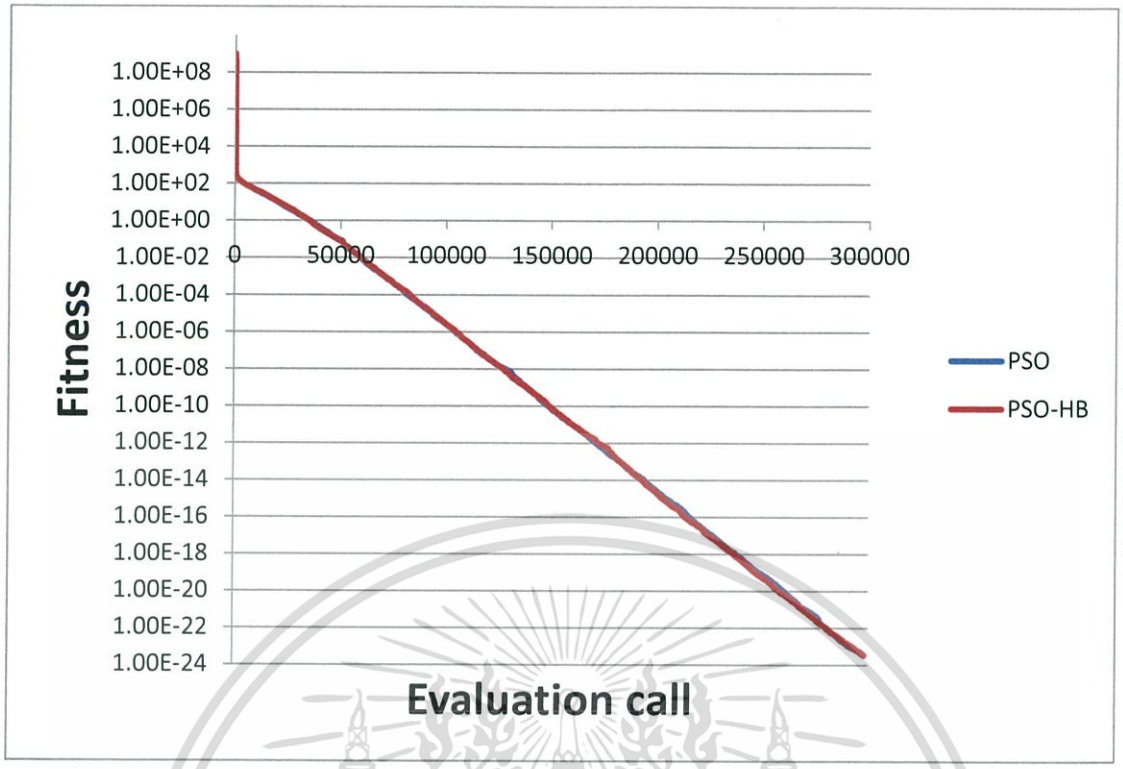
รูป 5.8 กราฟค่าความเหมาะสมในฟังก์ชัน Schaffer

5.6.2 Zakharov Function



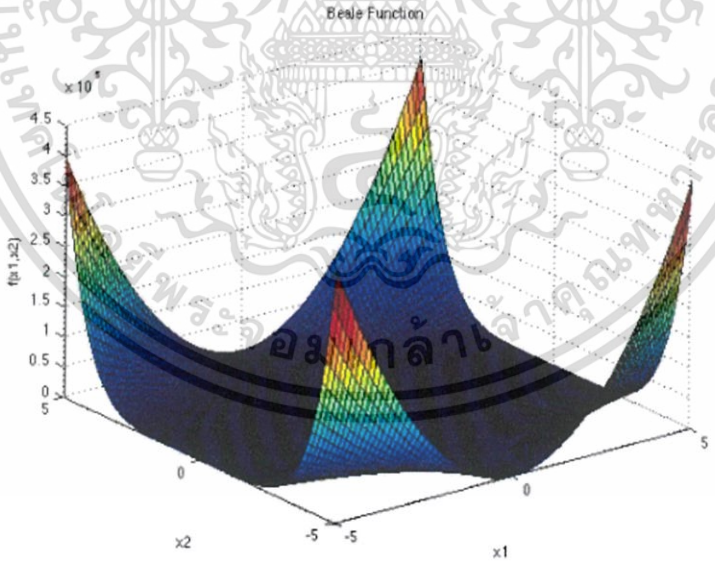
$$f(\mathbf{x}) = \sum_{i=1}^{d} x_i^2 + \left(\sum_{i=1}^{d} 0.5ix_i\right)^2 + \left(\sum_{i=1}^{d} 0.5ix_i\right)^4$$

รูป 5.9 Zakharov Function



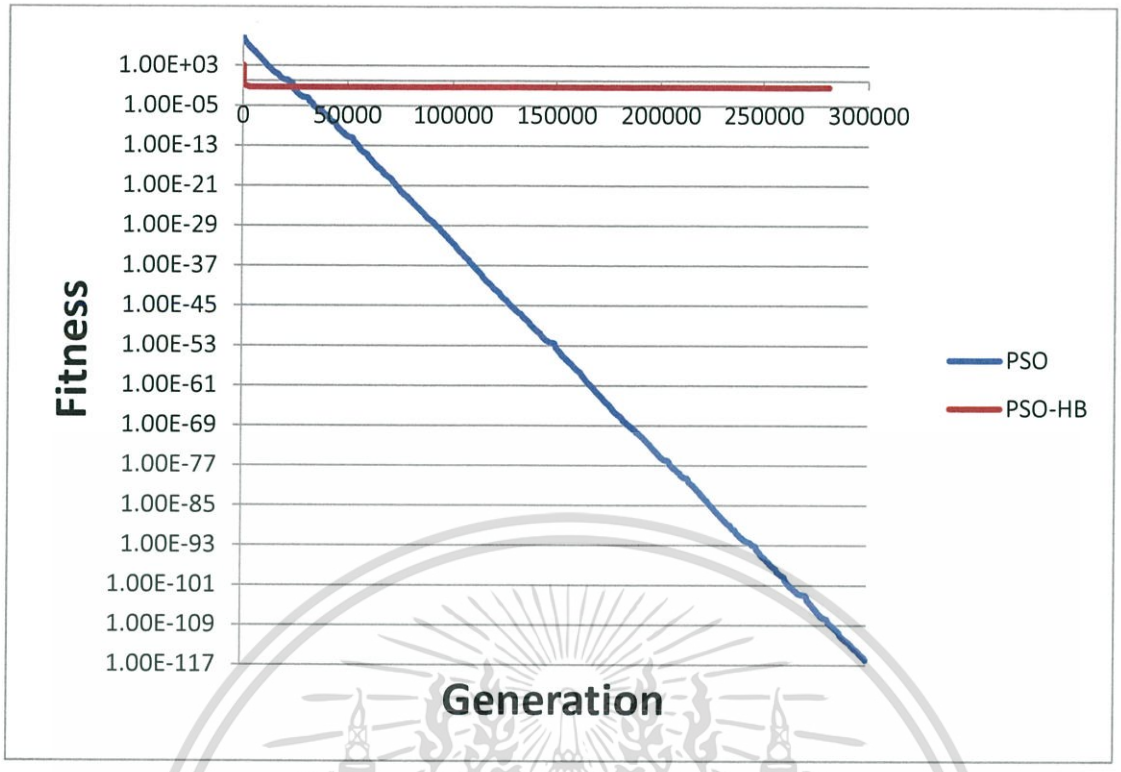
รูป 5.10 กราฟค่าความเหมาะสมในฟังก์ชัน Zakharov

5.6.3 Beale Function



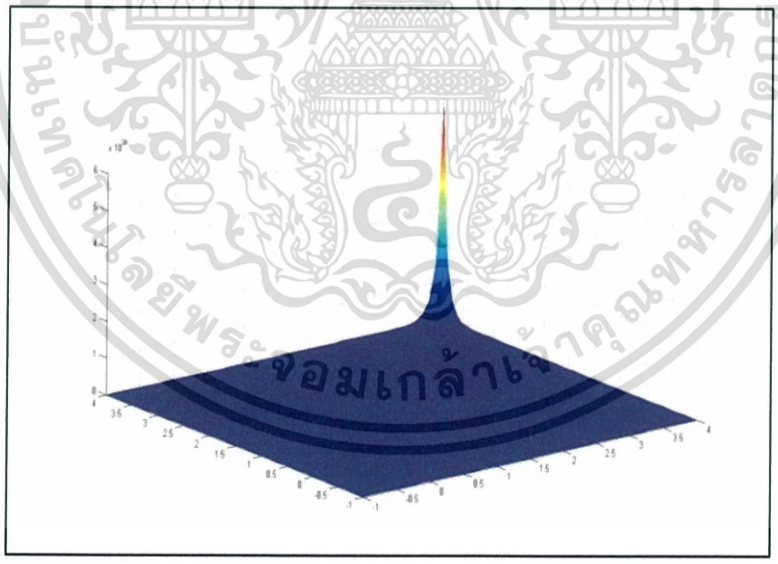
$$f(\mathbf{x}) = (1.5 - x_1 + x_1x_2)^2 + (2.25 - x_1 + x_1x_2^2)^2 + (2.625 - x_1 + x_1x_2^3)^2$$

รูป 5.11 Beale Function

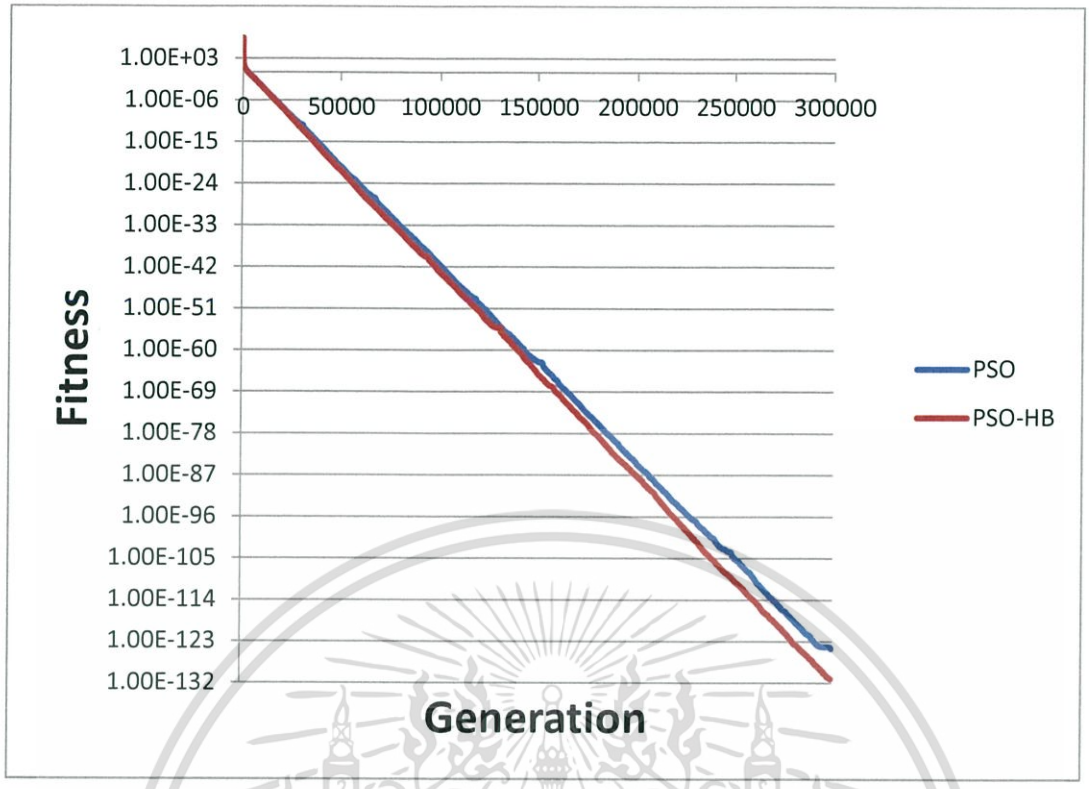


รูป 5.12 กราฟค่าความเหมาะสมในฟังก์ชัน Beale

5.6.4 Brown Function

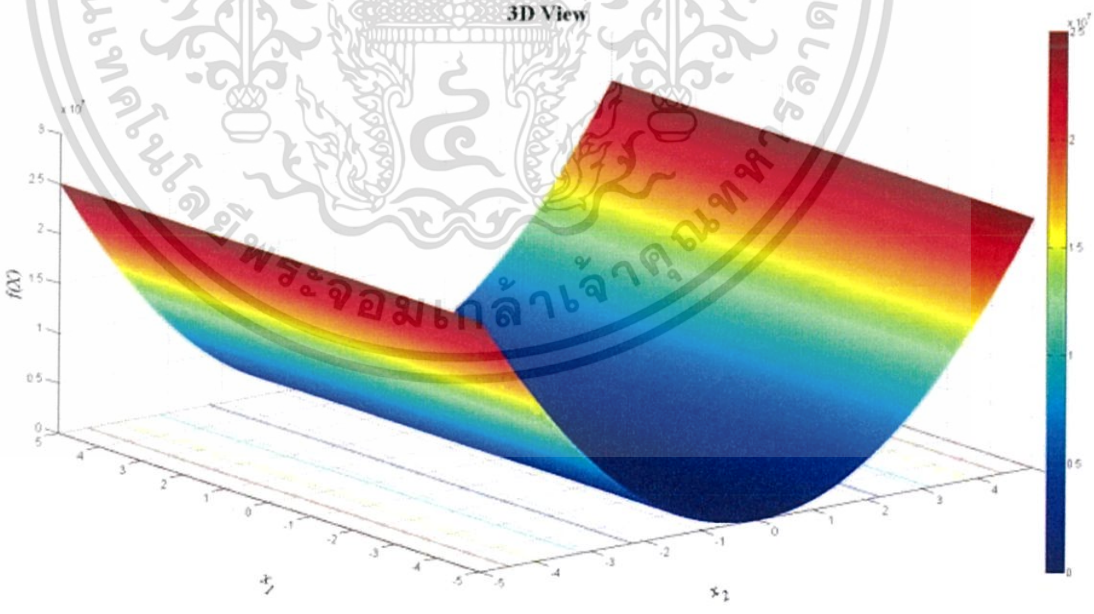


รูป 5.13 Brown Function

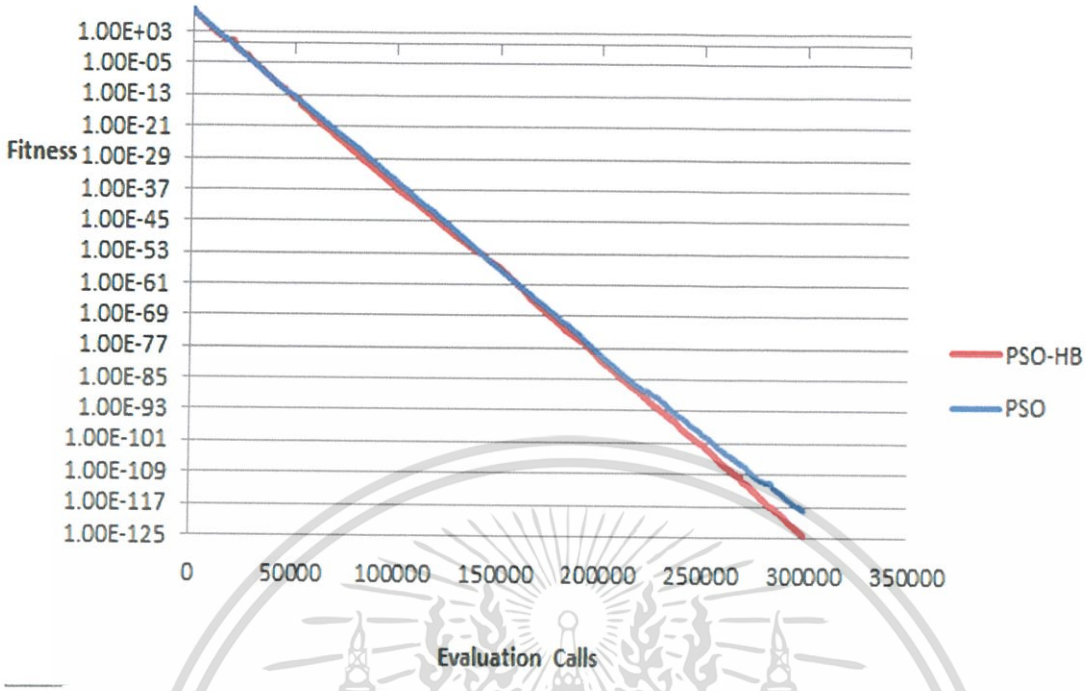


รูป 5.14 กราฟค่าความเหมาะสมในฟังก์ชัน Brown

5.6.5 Cigar Function

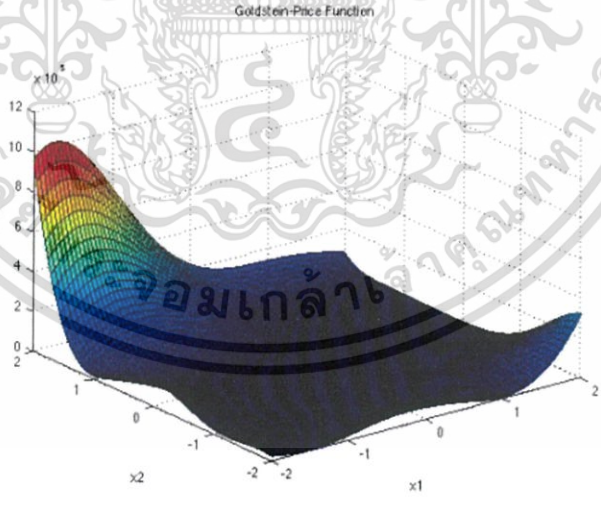


รูป 5.15 Cigar Function



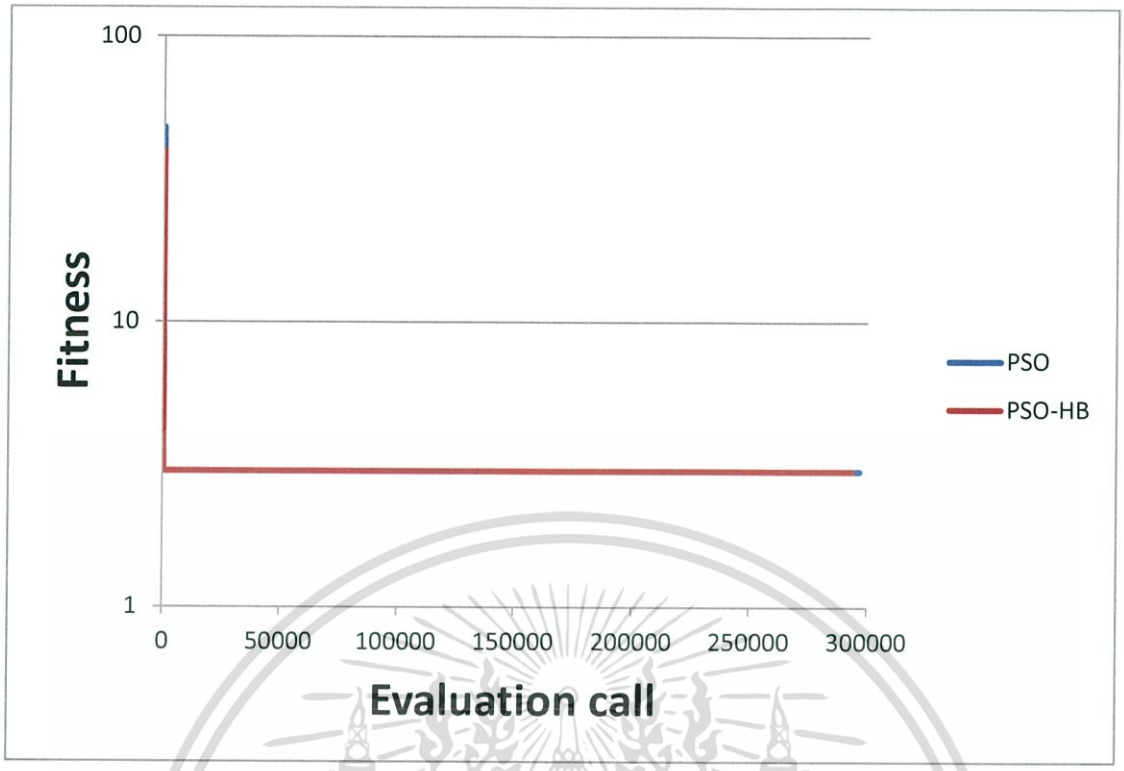
รูป 5.16 กราฟค่าความเหมาะสมในฟังก์ชัน Cigar

5.6.6 Goldstein-Price Function



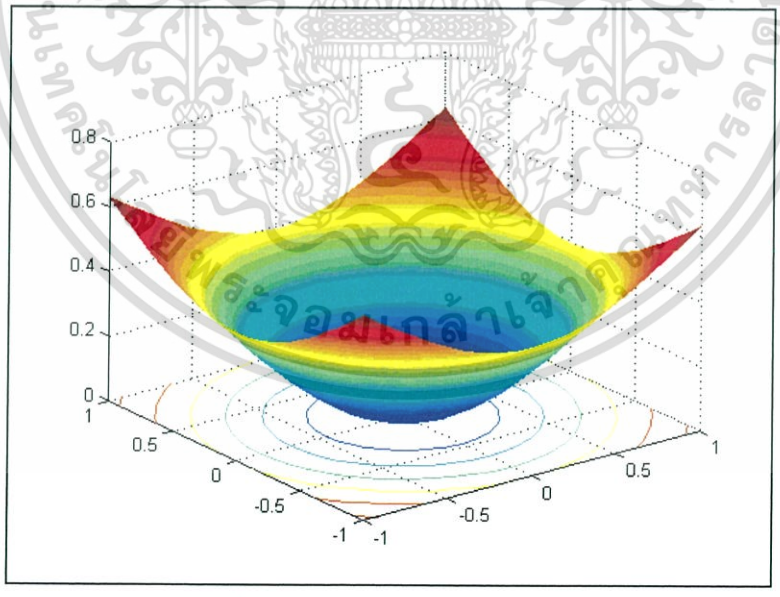
$$f(\mathbf{x}) = \left[1 + (x_1 + x_2 + 1)^2(19 - 14x_1 + 3x_1^2 - 14x_2 + 6x_1x_2 + 3x_2^2)\right] \times \left[30 + (2x_1 - 3x_2)^2(18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2)\right]$$

รูป 5.17 Goldstein-Price Function

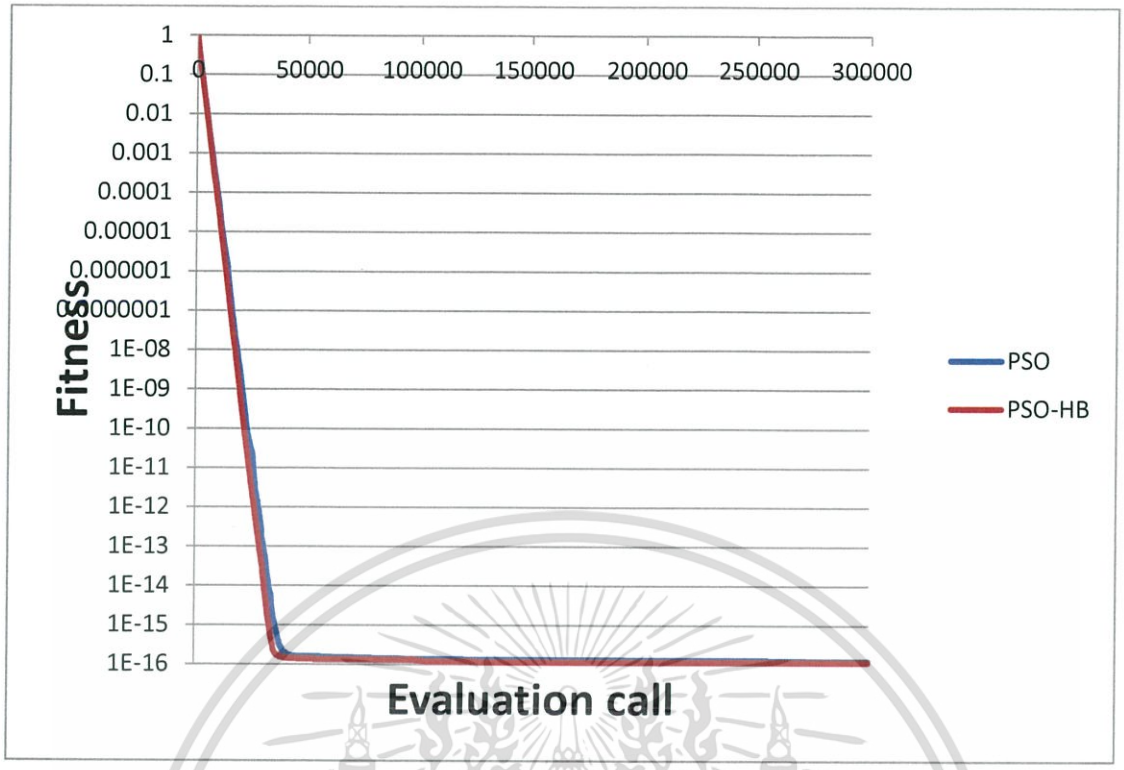


รูป 5.18 กราฟค่าความเหมาะสมในฟังก์ชัน Goldstein-Price

5.6.7 Exponential Function

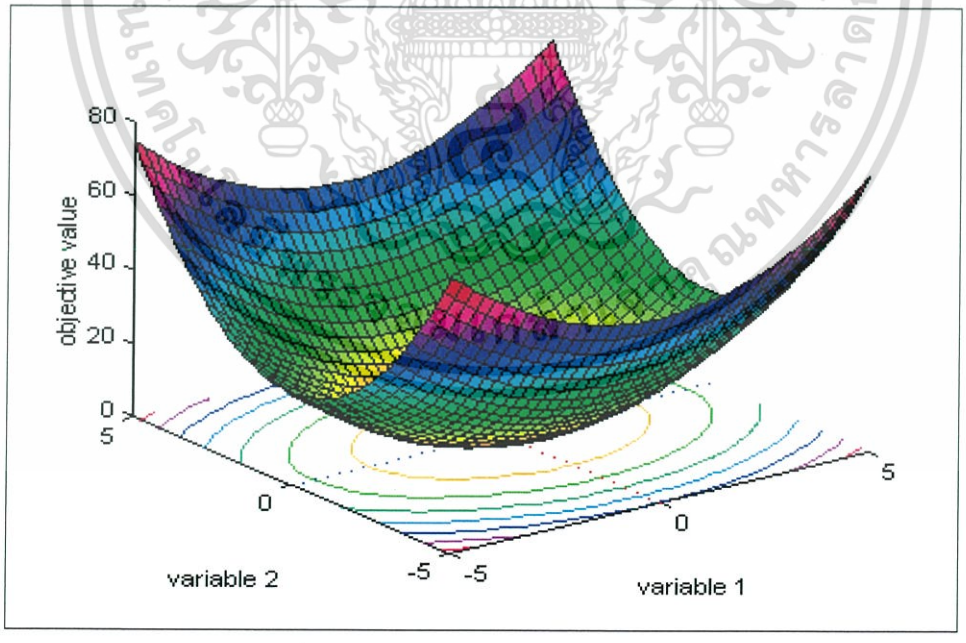


รูป 5.19 Exponential Function

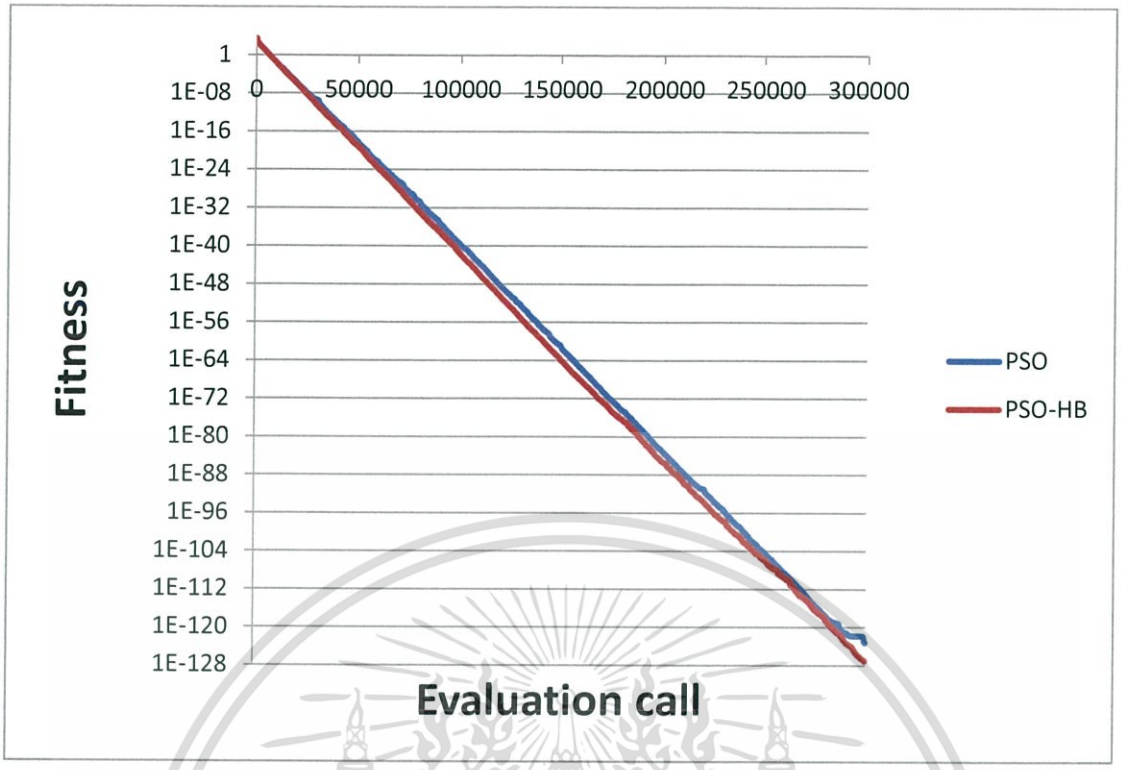


รูป 5.20 กราฟค่าความเหมาะสมในฟังก์ชัน Exponential

5.6.8 Parallel Ellipsoid Function

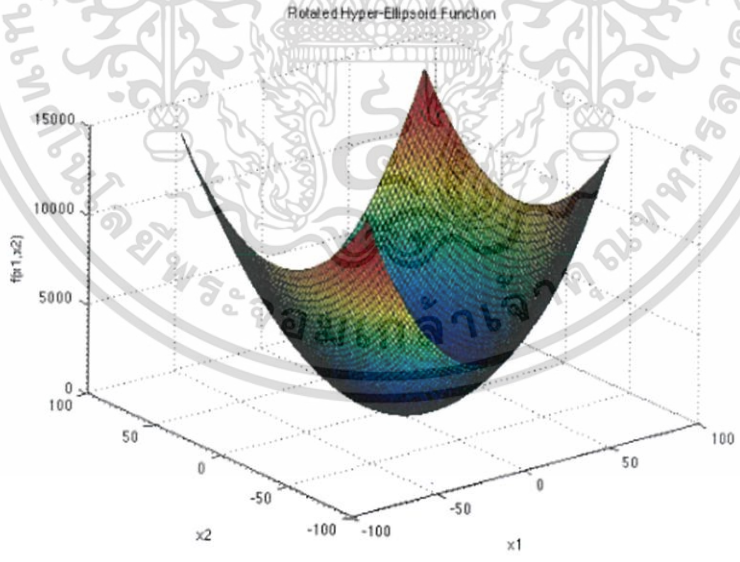


รูป 5.21 Parallel Ellipsoid Function



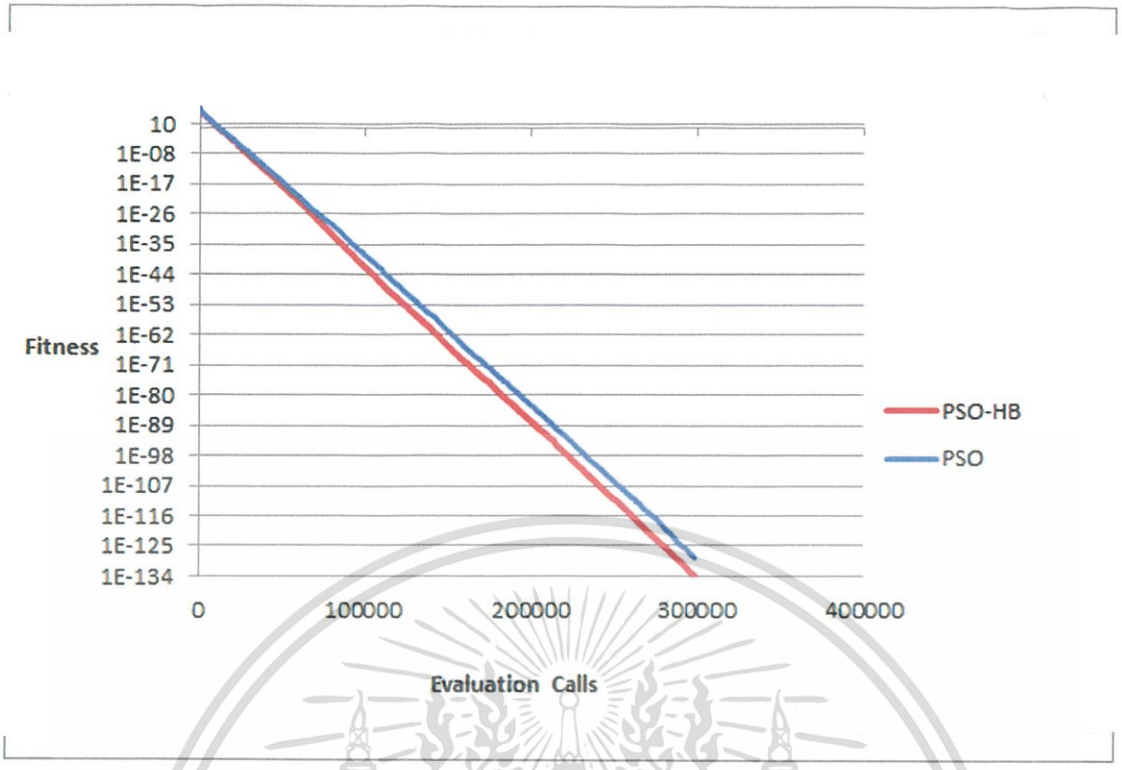
รูป 5.22 กราฟค่าความเหมาะสมในฟังก์ชัน Parallel Ellipsoid

5.6.9 Rotated Ellipsoid Function



$$f(\mathbf{x}) = \sum_{i=1}^{d} \sum_{j=1}^{i} x_j^2$$

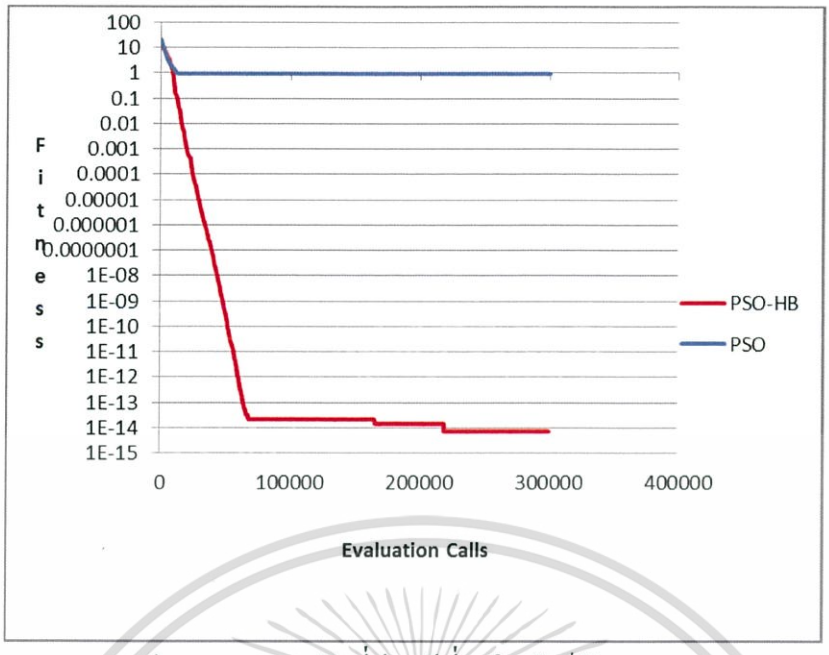
รูป 5.23 Rotated Ellipsoid Function



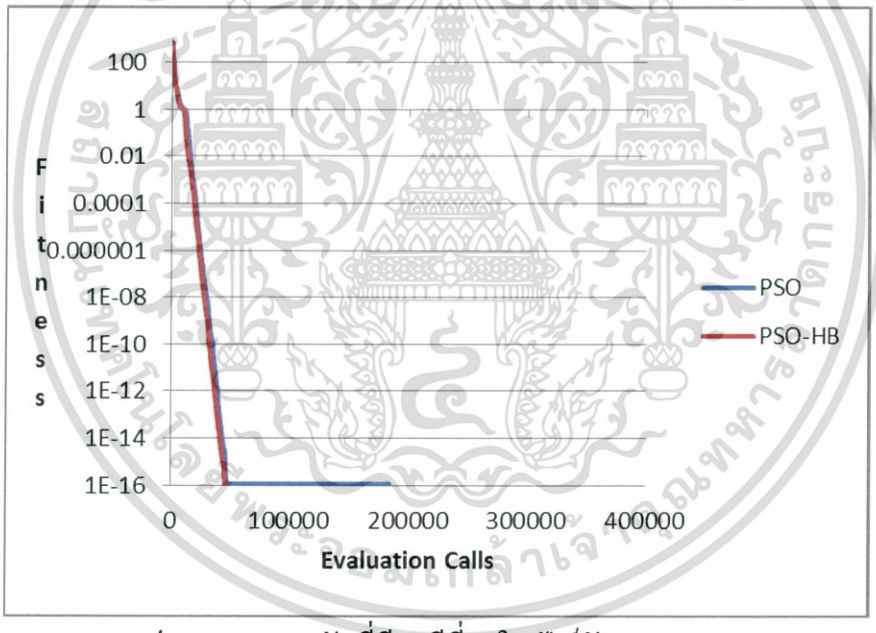
รูป 5.24 กราฟค่าความเหมาะสมในฟังก์ชัน Rotated Ellipsoid

5.7 ผลการทดสอบโดยวัดครั้งที่ดีที่สุด

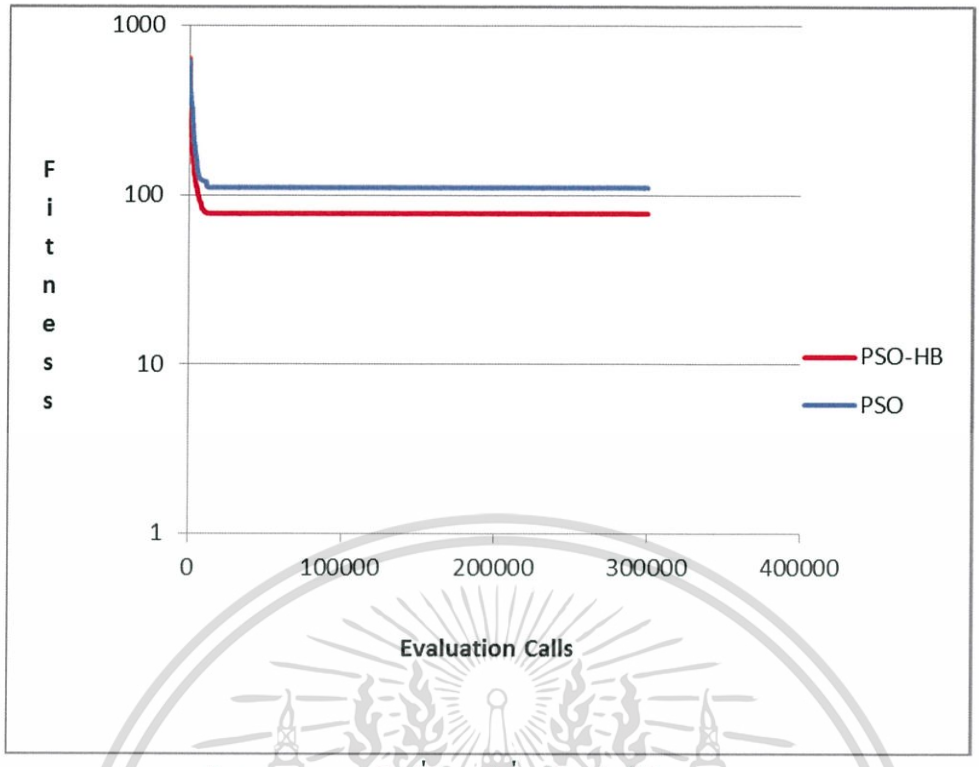
หัวข้อนี้นำเสนอผลการทดลองที่ดีที่สุดในการทดลอง 10 ครั้ง ของแต่ละฟังก์ชันทดสอบ การทดสอบรูปแบบค่าเฉลี่ยนั้นอาจทำให้เห็นผลได้ไม่ชัดเนื่องจากว่าการติดโลคอลออพติมัมเพียงแค่ครั้งเดียวจากการรันทั้งหมดก็ทำให้ค่าเฉลี่ยที่ได้มาแย่ลงอย่างมหาศาล การทดลองนี้จึงทดสอบละคัดเลือดรอบที่มีผลดีที่สุดมาแสดง



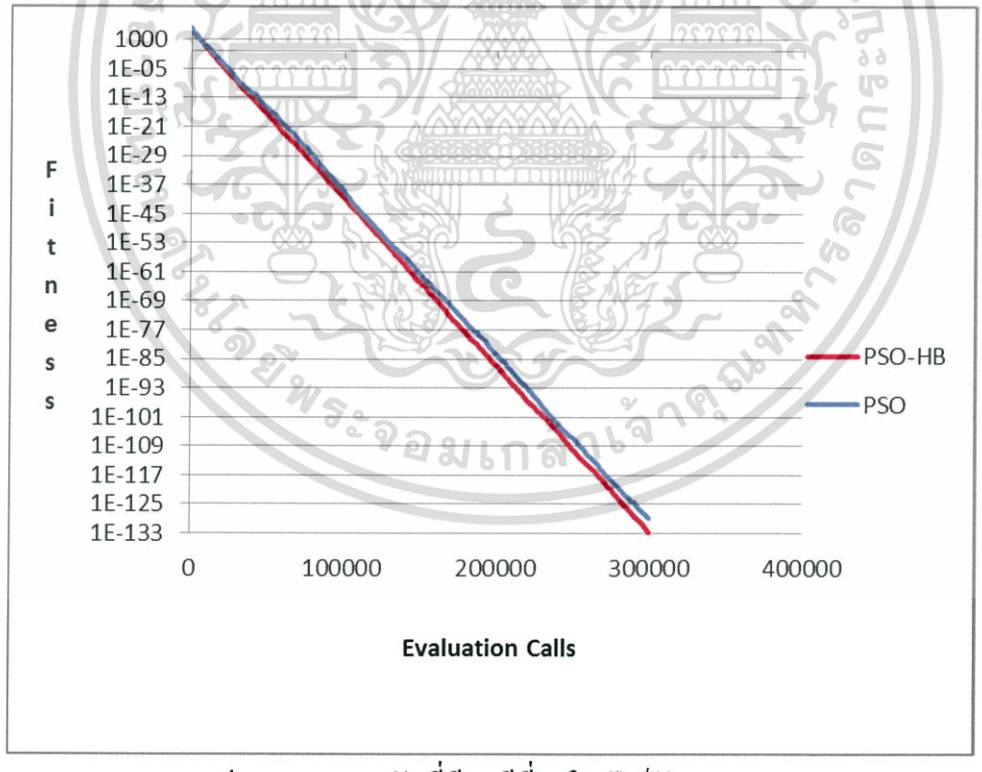
รูป 5.25 รอบการรันที่มีผลดีที่สุดในฟังก์ชัน Ackley



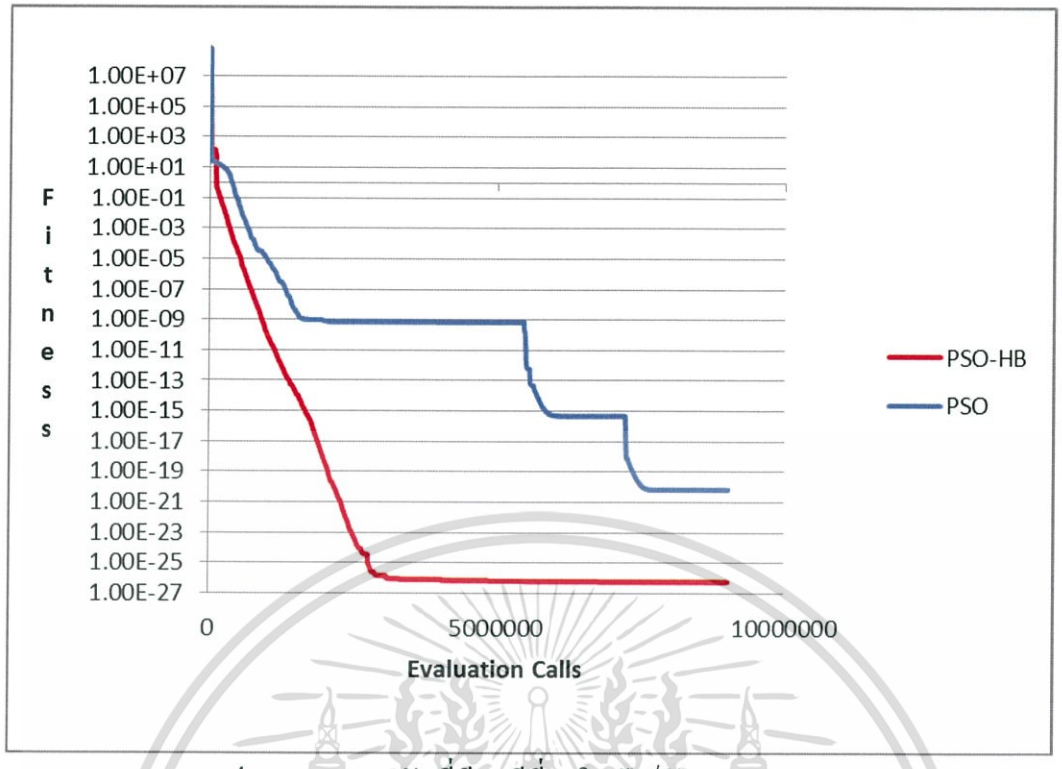
รูป 5.26 รอบการรันที่มีผลดีที่สุดในฟังก์ชัน Griewank



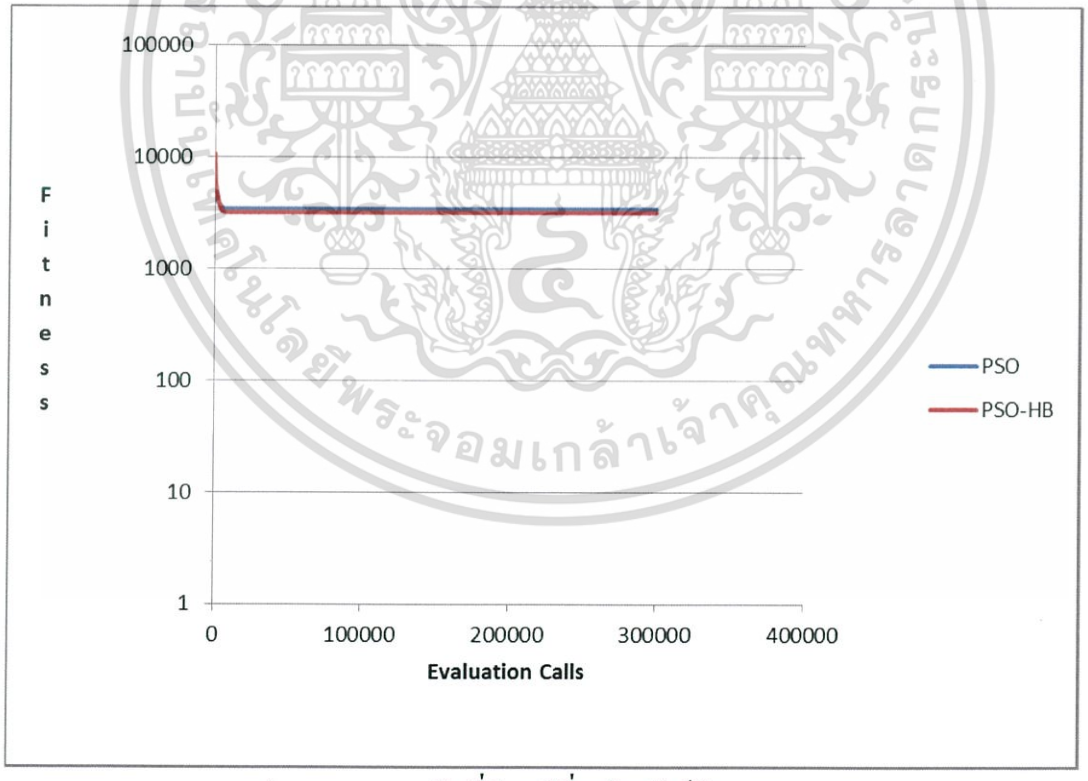
รูป 5.27 รอบการรันที่มีผลดีที่สุดในฟังก์ชัน Rastrigin



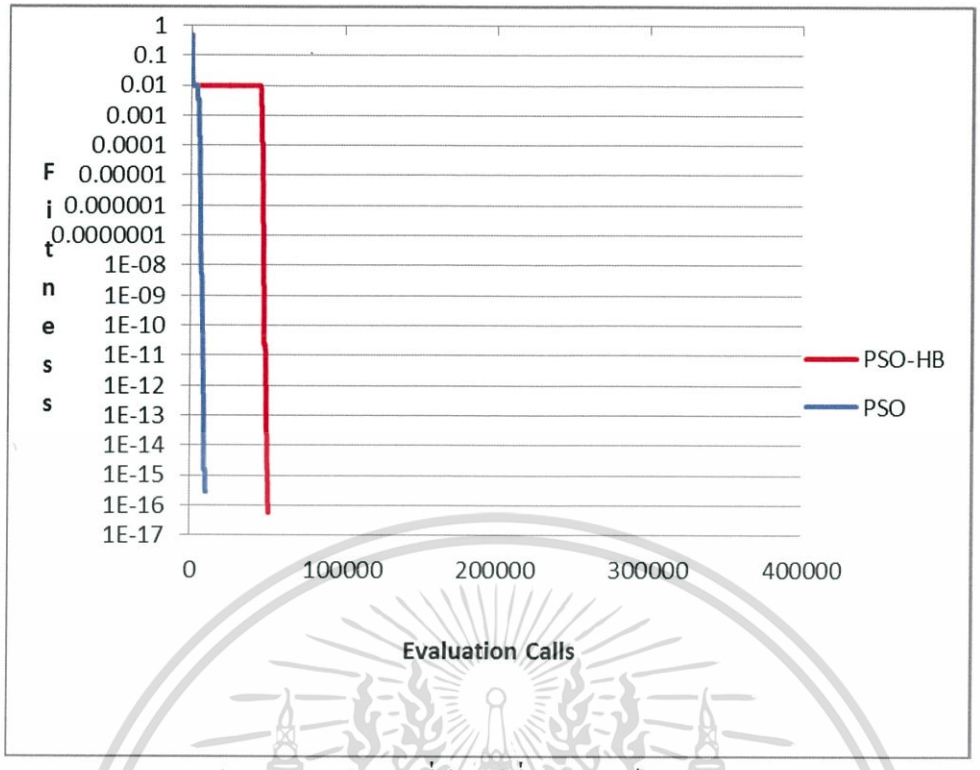
รูป 5.28 รอบการรันที่มีผลดีที่สุดในฟังก์ชัน Sphere



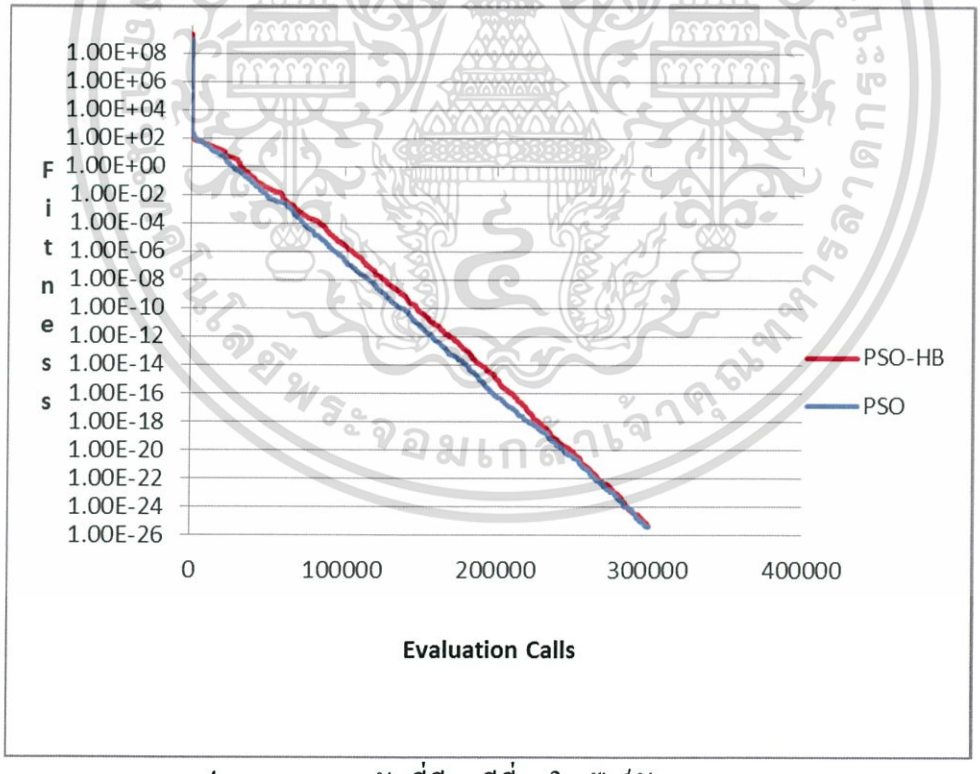
รูป 5.29 รอบการรันที่มีผลดีที่สุดในฟังก์ชัน Rosenbrock



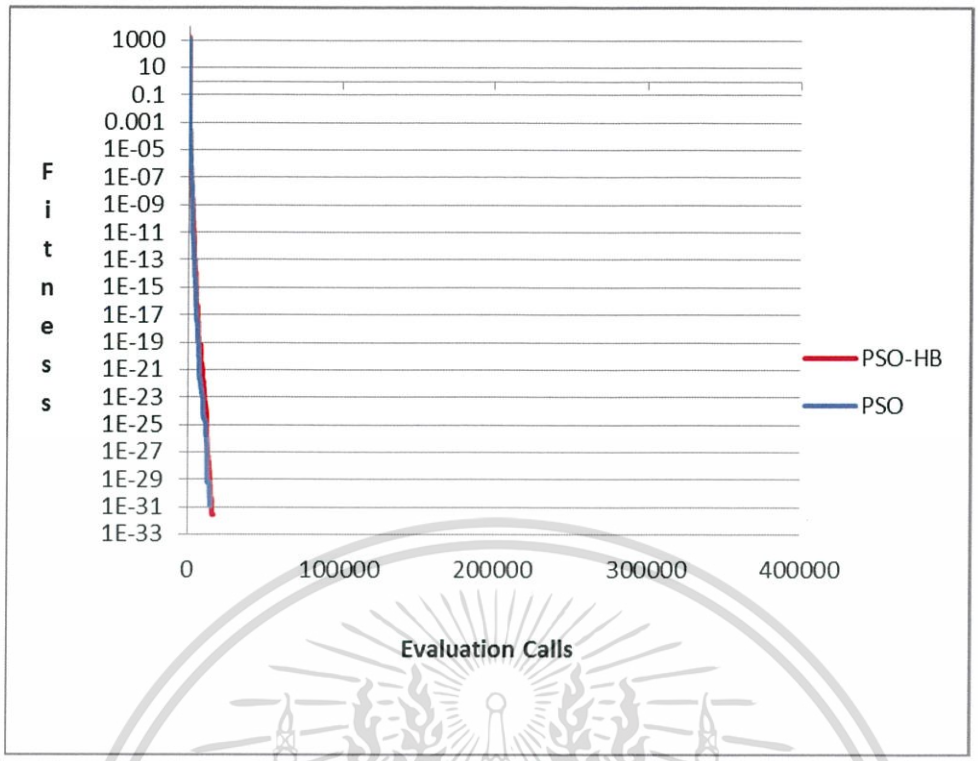
รูป 5.30 รอบการรันที่มีผลดีที่สุดในฟังก์ชัน Schwefel



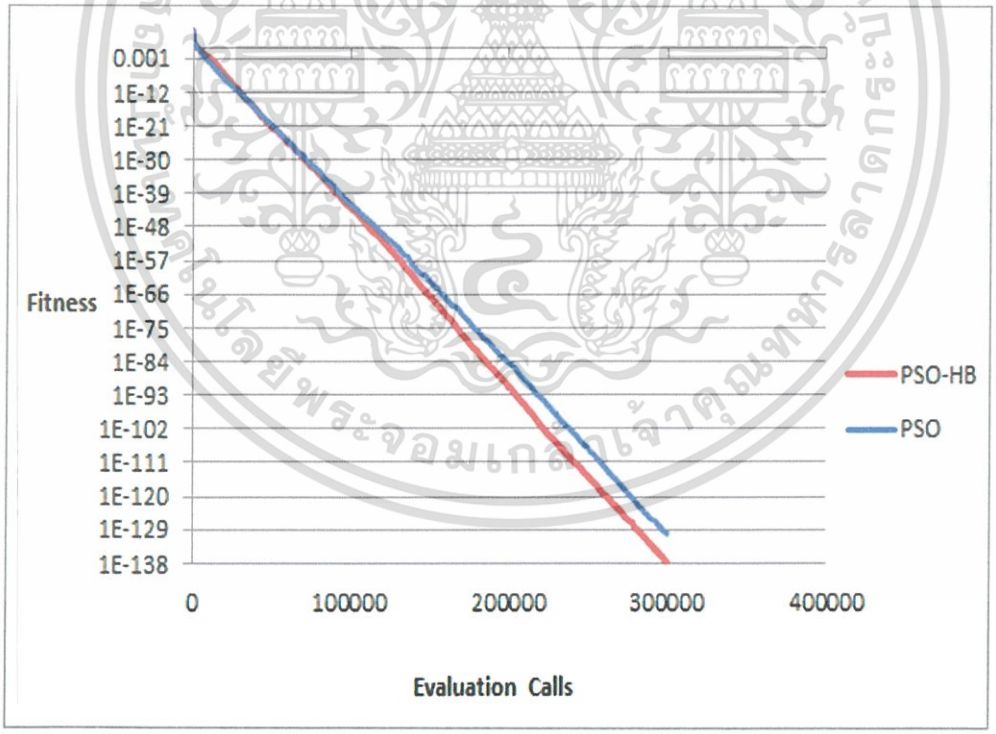
รูป 5.31 รอบการรันที่มีผลดีที่สุดในฟังก์ชัน Schaffer



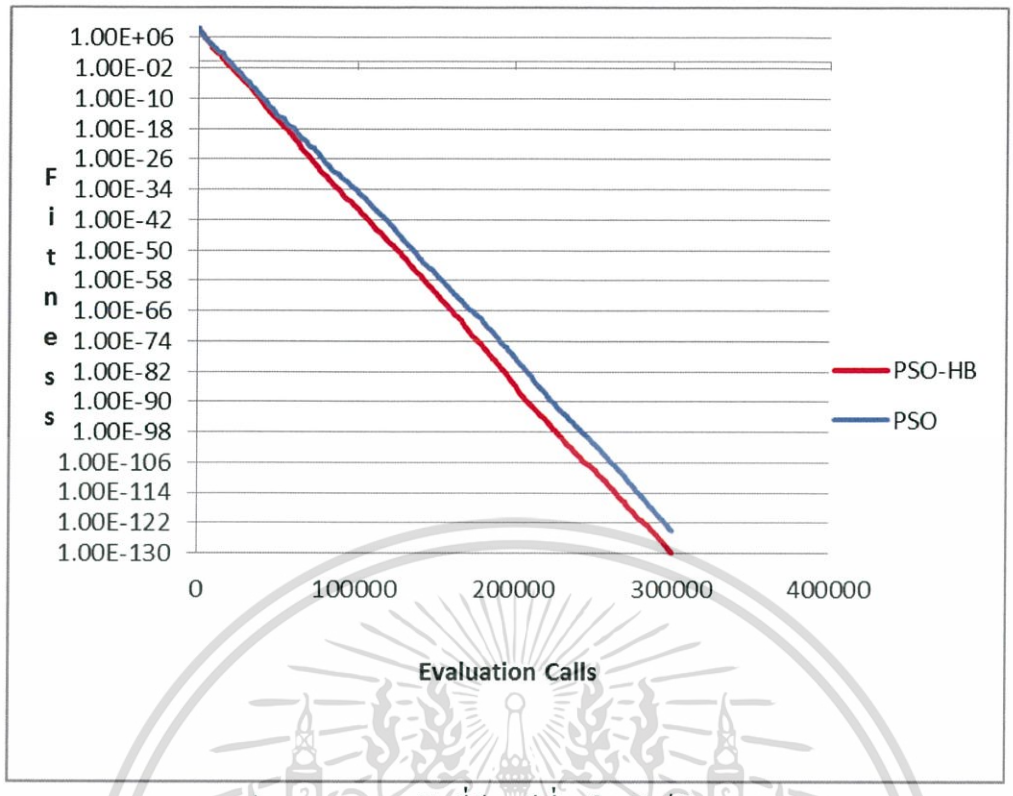
รูป 5.32 รอบการรันที่มีผลดีที่สุดในฟังก์ชัน Zakharov



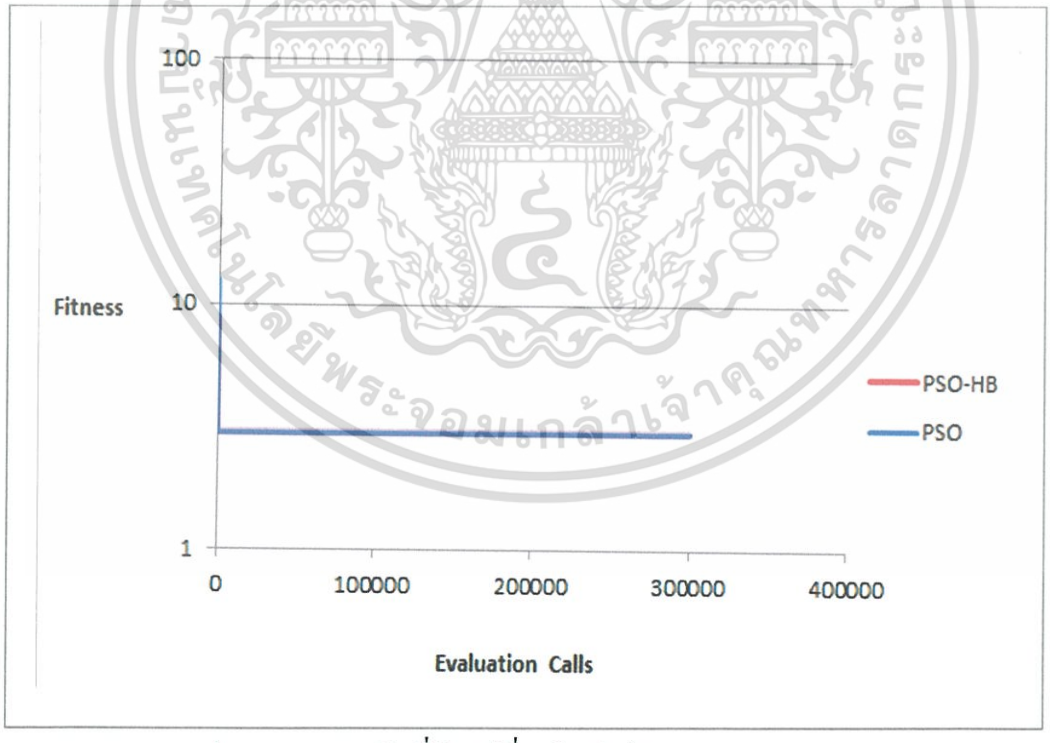
รูป 5.33 รอบการรันที่มีผลดีที่สุดในฟังก์ชัน Beale



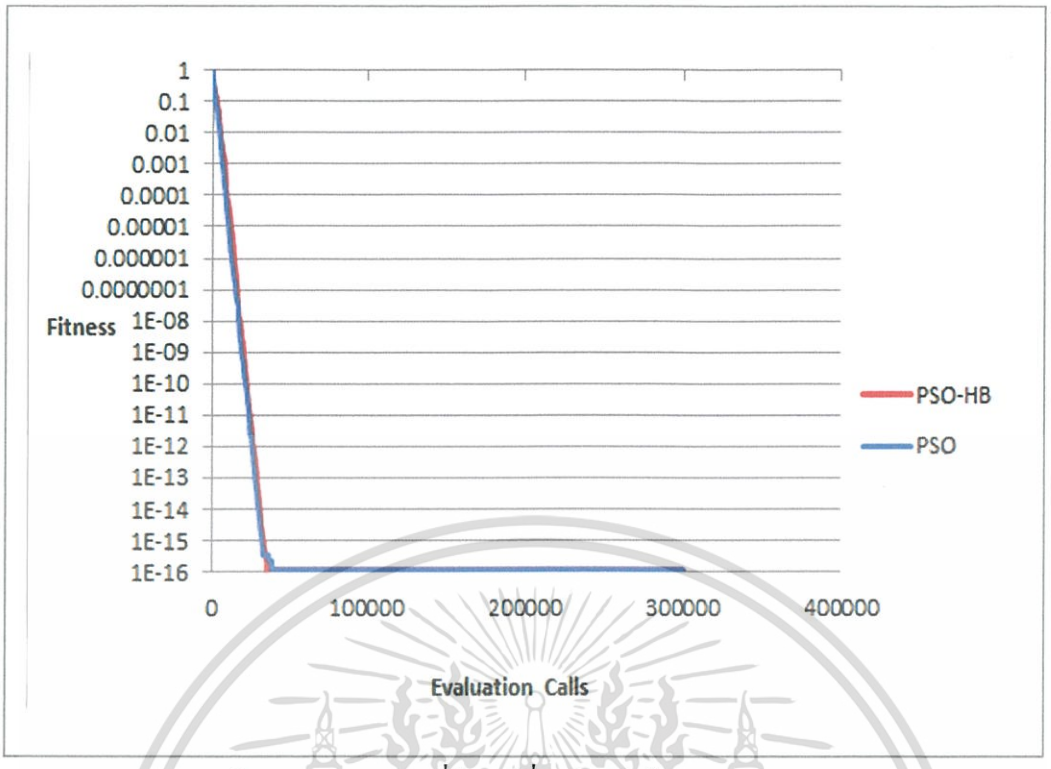
รูป 5.34 รอบการรันที่มีผลดีที่สุดในฟังก์ชัน Brown



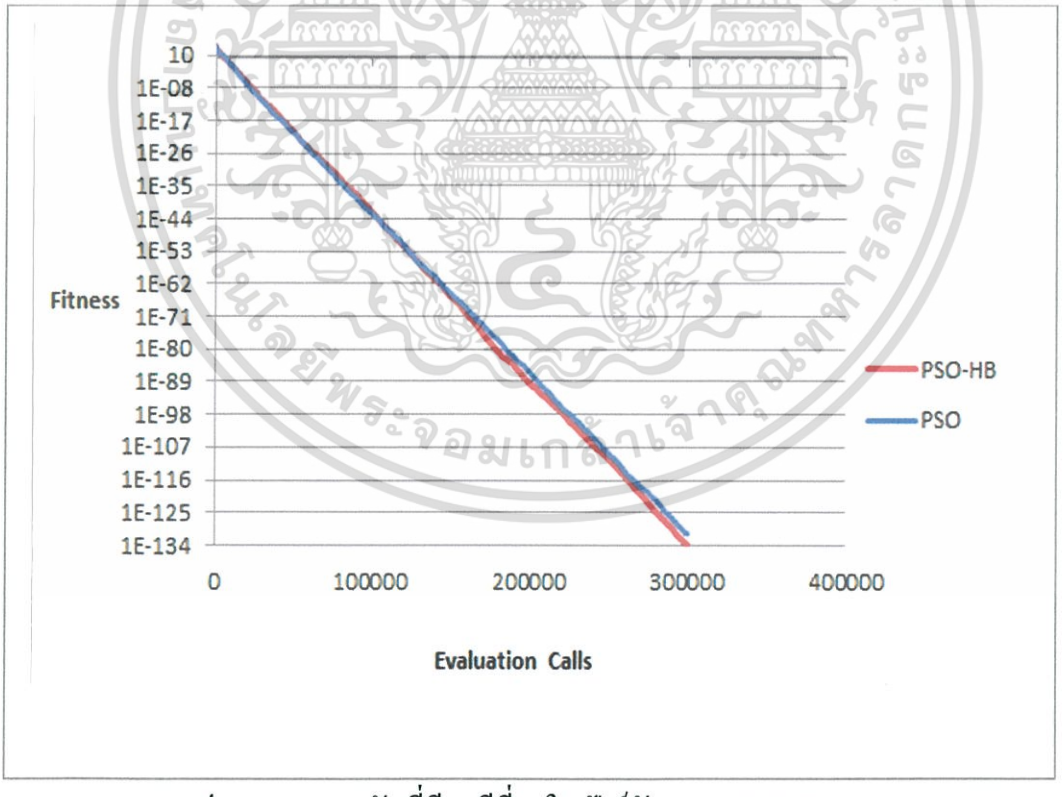
รูป 5.35 รอบการรันที่มีผลดีที่สุดในฟังก์ชัน Cigar



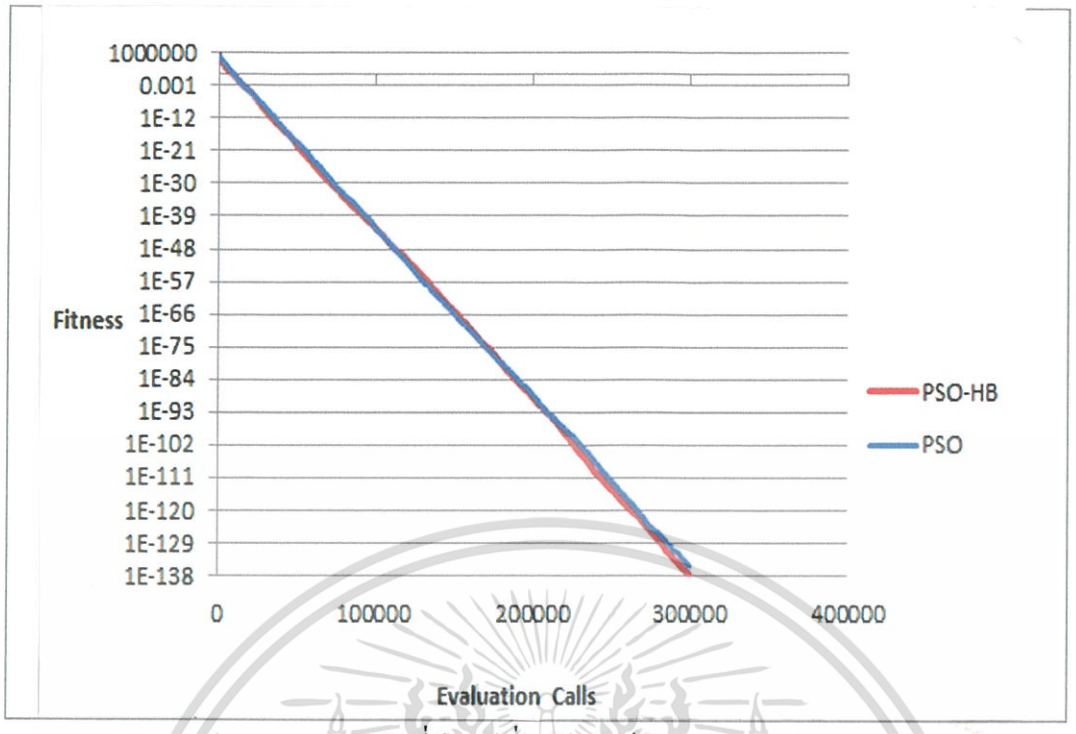
รูป 5.36 รอบการรันที่มีผลดีที่สุดในฟังก์ชัน Goldstein-Price



รูป 5.37 รอบการรันที่มีผลดีที่สุดในฟังก์ชัน Exponential



รูป 5.38 รอบการรันที่มีผลดีที่สุดในฟังก์ชัน Parallel Ellipsoid



รูป 5.39 รอบการรันที่มีผลดีที่สุดในฟังก์ชัน Rotated Ellipsoid

5.8 เปรียบเทียบผลการทดลอง

การเปรียบเทียบผลการทดลองในหัวข้อนี้ จะเปรียบเทียบทั้งหมด 2 รูปแบบ โดยแบบแรกทำการรันทั้งหมด 10 ครั้ง และนำรอบการรันที่ที่ที่สุดเปรียบเทียบ โดยนำค่าความเหมาะสมที่ดีที่สุดเมื่อฝูงอนุภาคไม่สามารถหาค่าที่ดีกว่านี้ได้อีกแล้วหรือเรียกสั้นๆว่าการตายของฝูงนกมาใช้ในการเปรียบเทียบความสามารถระหว่าง PSO และ PSO-HB นอกจากค่าความเหมาะสมที่ดีที่สุดแล้วเรานำค่าจำนวนการประมวลผล (evaluation call) ของการพบครั้งแรกของคำตอบที่ดีที่สุดที่ฝูงนกหาได้ก่อนจะตายนี้ ดังแสดงในตารางที่ 5.6

ตารางที่ 5.6 ตารางเปรียบเทียบค่าเหมาะสมที่ดีที่สุด

Function		PSO	PSO-HB
Ackley	Fitness	0.931305467	7.11E-15
	Evacall	23710	218866
Griewank	Fitness	0	0
	Evacall	183563	45587
Rastrigin	Fitness	110.4400489	77.60656498
	Evacall	23019	25718
Sphere	Fitness	1.50E-129	1.72E-133
	Evacall	300011	300047
Rosenbrock	Fitness	6.83E-21	5.87E-27
	Evacall	7785186	8855561
Schwefel	Evacall	3436.229488	2896.637677
	Fitness	15155	4143
Schaffer	Evacall	0	0
	Fitness	10936	51438
Zakharov	Evacall	4.15E-26	5.00E-26
	Fitness	299168	299747
Beale	Evacall	0	0
	Fitness	14717	16850
Cigar	Evacall	9.21E-125	1.31E-130
	Evacall	299966	299977
Brown	Fitness	7.89E-131	2.31E-138
	Evacall	299964	299990
Goldstein-Price	Fitness	3	3
	Evacall	7084	7790
Exponential	Fitness	1.11E-16	1.11E-16
	Evacall	39113	36214
Parallel Ellipsoid	Fitness	1.83E-131	1.95E-134
	Evacall	299987	299950
Rotated Ellipsoid	Evacall	4.58E-136	2.29E-138
	Fitness	299951	299990

ในส่วนการเปรียบเทียบที่สองจะนำค่าความเหมาะสมที่ดีที่สุดเมื่อฝูงนกตายจากการรันทั้ง 10 ครั้งมาเฉลี่ยแล้วเปรียบเทียบ คู่กันกับค่าเฉลี่ยของจำนวนการประเมินผลที่เจอค่าความเหมาะสมนี้ครั้งแรกจากการรันทั้งหมด 10 ครั้ง

ตาราง 5.8 ตารางเปรียบเทียบค่าเหมาะสมที่ดีที่สุดโดยเฉลี่ย

Function		PSO	PSO-HB
Ackley	Fitness	1.795254486	2.68E-01
	Evacall	24946.2	38143.7
Griewank	Fitness	0.026218499	0.021611482
	Evacall	27640	25615
Rastrigin	Fitness	148.6463463	133.0255493
	Evacall	31523.7	25579.7
Sphere	Fitness	7.03E-123	1.47E-128
	Evacall	299894.5	299736.8
Rosenbrock	Fitness	1.725457499	0.3986625
	Evacall	2130262.9	1614519.1
Schwefel	Evacall	3678.279464	3553.437291
	Fitness	17059.8	54034.4
Schaffer	Evacall	0.002914933	9.72E-04
	Fitness	43671.4	162159.8
Zakharov	Evacall	4.65E-24	5.53E-24
	Fitness	298497.9	297331.8
Beale	Evacall	0.076207123	0.075675658
	Fitness	2844.9	12634.7
Cigar	Evacall	7.47E-119	4.60E-125
	Evacall	299868.6	299689.7
Brown	Fitness	1.07E-125	3.01E-134
	Evacall	299685	299164.5
Goldstein-Price	Fitness	3	3
	Evacall	7011	7131.3
Exponential	Fitness	1.11E-16	1.11E-16
	Evacall	260747	191895.9
Parallel Ellipsoid	Fitness	5.55E-124	3.58E-131
	Evacall	299817	299932.4
Rotated Ellipsoid	Evacall	2.84E-129	1.36E-134
	Fitness	299114.6	299143.7

บทที่ 6

สรุปผลการทดลองและข้อเสนอแนะ

6.1 สรุปผลการทดลอง

การค้นหาคำตอบแบบกลุ่มอนุภาค (Particle Swarm Optimization) เป็นการค้นหาค่าเหมาะสมที่สุดของฟังก์ชั่นที่เลียนแบบการเคลื่อนที่ของฝูงนกที่บินตามจุดที่เหมาะสมที่สุดของตัวเอง (PBest) และจุดที่เหมาะสมที่สุดของจ่าฝูง (GBest) โดยเมื่อเวลาผ่านไป ตัวนกเองจะค่อยๆเข้าใกล้กันมากขึ้นเรื่อยๆ ทำให้บริเวณของการค้นหาแคบลงเรื่อยๆ จนสุดท้ายอาจจะติดอยู่ในจุดสุดสัมพัทธ์ (Local Optima) ที่ไม่ได้เป็นจุดสุดสัมบูรณ์ (Global Optima) ของฟังก์ชั่นได้ เพื่อแก้ปัญหานั้น โครงงานฉบับนี้จึงได้ทำการเพิ่มบริเวณของการค้นหาด้วยการสุ่มคำตอบเพิ่มจากกลุ่มประชากร (GBest Pool) เพราะค่าตำแหน่งของนกจ่าฝูงที่เคยผ่านมาอาจมีค่าตำแหน่งในบางมิติที่เป็นค่าที่ดี ทำให้พบคำตอบที่ดีกว่า และหลุดจากจุดสุดสัมพัทธ์ได้

สิ่งที่การค้นหาคำตอบแบบกลุ่มอนุภาคแบบมีกลุ่มประชากรแตกต่างจากการค้นหาคำตอบแบบกลุ่มอนุภาคแบบธรรมดาคือ ในทุกครั้งที่เคลื่อนไหวแต่ละอนุภาค จะมีการเก็บตำแหน่งที่ได้ผ่านมาของแต่ละอนุภาค เพิ่มลงในกลุ่มประชากรโดยที่หากคำตอบที่ได้ดีกว่าค่าที่ดีที่สุดของจ่าฝูง ก็จะแทนที่ค่าที่ดีที่สุดของจ่าฝูงเลย แต่ถ้ายังไม่ดี ก็จะเปรียบเทียบกับค่าอื่นๆในกลุ่มประชากรต่อไป และใส่แทรกลงก่อนหน้าประชากรที่แย่กว่าตัวมันและดันตัวที่แย่กว่าลงไปในลำดับถัดๆไป และในแต่ละรอบที่เคลื่อนไหวอนุภาคครบทุกอนุภาค จะทำการสร้างคำตอบเพิ่มจากกลุ่มประชากร และใส่ลงในกลุ่มประชากรเช่นกัน

ในบางช่วงค่าความเหมาะสมที่คำตอบอยู่ใกล้กับ local optima นั้นการใช้วิธีการปรับปรุงขั้นตอนวิธีหาค่าเหมาะสมอย่างกลุ่ม(PSO-HB)จะสามารถช่วยให้คำตอบหลุดจาก global local optimum(Global Best) ด้วยการสร้างประชากรใหม่อย่างสุ่มและทำให้เกิดความหลากหลายมากขึ้นกว่าขั้นตอนวิธีหาค่าเหมาะสมอย่างกลุ่ม(PSO) ในบางช่วงอาจพบว่าความหลากหลายจากการทดสอบด้วย benchmark function ของ PSO-HB มีค่าต่ำกว่า PSO ซึ่งเป็นเพราะค่าคำตอบของ PSO-HB มีค่าน้อยกว่าในขณะนั้นๆ และถ้าเทียบค่าความหลากหลายต่อความละเอียดของคำตอบ โดยส่วนใหญ่เราพบว่า PSO-HB มีค่าความหลากหลายมากกว่า PSO

ปัญหาที่พบในการทำ PSO-HB เราพบว่าในกรณีที่ฝูงนกเกิดการเคลื่อนตัวจนไปเรื่อยๆและติดใน local optima ในท้ายที่สุดแล้วกลุ่มอดีตค่าที่ดีที่สุดนั้นไม่อาจคงความหลากหลายไว้ได้ซึ่งส่งผลให้คำตอบที่สร้างขึ้นจากกลุ่มอดีตค่าที่ดีสุดไม่ดีไปกว่าค่า Gbest ได้อีก

6.2 ข้อเสนอแนะ

วิธีการปรับปรุงขั้นตอนวิธีหาค่าเหมาะสมที่สุดอย่างกลุ่มสามารถช่วยหลีกหนี local optima ได้ดังจะเห็นได้จากผลการทดลอง แต่ถึงอย่างไรก็ตามเปอร์เซ็นความสำเร็จที่ได้คำตอบที่ถูกต้องนั้นยังไม่ถึงร้อย เนื่องจากกลุ่มอดีตค่าที่ดีที่สุดนั้นไม่สามารถคงความหลากหลายไว้ได้ เมื่อประชากรของฝูงนกค่อยๆลู่เข้าสู่จุด global local optimum ด้วยแรงดูดจาก Gbest มากขึ้นจนความหลากหลายของประชากรน้อยลงเรื่อยๆ ส่งผลให้กลุ่มอดีตค่าที่ดีที่สุดถูกแทนที่ด้วยค่าที่ดีที่สุดค่าใหม่ที่นกในฝูงเจอซึ่งมีความหลากหลายน้อย เมื่อกลุ่มอดีตค่าที่ดีที่สุดถูกแทนด้วยค่าที่มีความหลากหลายน้อยมากๆเข้าก็จะส่งผลให้ความหลากหลายหมดตามฝูงนกไปและตายตามฝูงนกไปในที่สุด

จากปัญหาดังกล่าวนั้นในการทดลองนี้ได้พยายามหาวิธีการคัดเลือกประชากรที่จะเข้ามาแทนค่าในกลุ่มอดีตค่าที่ดีที่สุดเพื่อดำรงไว้ซึ่งความหลากหลาย แต่ก็ไม่เป็นผลสำเร็จ เนื่องจากวิธีการคัดกรองยังมีความชาญฉลาดไม่พอ และไม่ได้ส่งผลต่อประสิทธิภาพอย่างนัยยะสำคัญจึงไม่ได้นำเสนอวิธีคัดกรองอดีตค่าที่ดีที่สุดดังกล่าว

บรรณานุกรม

Bratton, Daniel, and James Kennedy. "Defining a standard for particle swarm optimization." 2007 IEEE swarm intelligence symposium. IEEE, 2007.

Shi, Yuhui, and Russell Eberhart. "A modified particle swarm optimizer." Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on. IEEE, 1998.

Eberhart, Russ C., and James Kennedy. "A new optimizer using particle swarm theory." Proceedings of the sixth international symposium on micro machine and human science. Vol. 1. 1995.

Shi, Yuhui. "Particle swarm optimization: developments, applications and resources." evolutionary computation, 2001. Proceedings of the 2001 Congress on. Vol. 1. IEEE, 2001.

Chelouah, Rachid, and Patrick Siarry. "A continuous genetic algorithm designed for the global optimization of multimodal functions." Journal of Heuristics 6.2 (2000): 191-213.

Jiang, Zheng, et al. "A hybrid genetic algorithm integrated with sequential linear programming." Machine Learning and Cybernetics, 2003 International Conference on. Vol. 2. IEEE, 2003.

Shi, Yuhui, and Russell C. Eberhart. "Empirical study of particle swarm optimization." Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on. Vol. 3. IEEE, 1999.

Holland, John H. "Outline for a logical theory of adaptive systems." Journal of the ACM (JACM) 9.3 (1962): 297-314.

Holland, John H. Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence. U Michigan Press, 1975.

Van Den Bergh, Frans. "An Analysis of Particle Swarm Optimizers (PSO)." (2001).

Riget, Jacques, and Jakob S. Vesterstrøm. "A diversity-guided particle swarm optimizer-the ARPSO." Dept. Comput. Sci., Univ. of Aarhus, Aarhus, Denmark, Tech. Rep 2 (2002): 2002.

Monson, Christopher K., and Kevin D. Seppi. "Exposing origin-seeking bias in PSO." Proceedings of the 7th

Gehlhaar, Daniel K., and David B. Fogel. "Tuning Evolutionary Programming for Conformationally Flexible

Eberhart, Russ C., and Yuhui Shi. "Comparing inertia weights and constriction factors in particle swarm

Simon Fraser University, 2016.**Optimization Test Problems**. [Online] Available:https://www.sfu.ca/~ssurjano/optimization.html

วิชญะ สมเกียรติ. 2558. “การปรับประชากรอย่างยืดหยุ่นสำหรับวิธีการประยุกต์การกลายพันธุ์และรีโพซิชันเข้ากับการหาค่าเหมาะสมแบบอนุภาคกลุ่ม.” วิทยานิพนธ์วิศวกรรมศาสตรมหาบัณฑิต สาขาวิชาวิศวกรรมคอมพิวเตอร์, สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง.

ภาคผนวก

ภาคผนวก ก

Source Code

ก.1 Package evaluateFunctions

ก.1.1 Abstract Class FitnessFunction

```
package evaluateFunctions;
public abstract class FitnessFunction
{
	public int callCount;
	public abstract double getMaxValue();
	public abstract double getMinValue();
	public abstract double getMaxSpawnPoint();
	public abstract double getMinSpawnPoint();
	public abstract double getOffset();
	public double execute(double[] x)
	{
		callCount = callCount + 1;
		return 0;
	}
}
```

ก.1.2 Class Ackley

```
package evaluateFunctions;
public class Ackley extends FitnessFunction{
    private static final double MAXVALUE = 32;
    private static final double MINVALUE = -32;
    private double offset = 0;

    public Ackley() {

    }
    public Ackley(double offset) {
        this.offset = offset;
    }
    @Override
    public double getMaxValue() {
        return MAXVALUE;
    }

    @Override
    public double getMinValue() {
        return MINVALUE;
    }

    @Override
    public double execute(double[] x) {
```

```
        super.execute(x);
         double sum1 = 0.0;
         double sum2 = 0.0;
         for (int i = 0; i < x.length; i++) {// x.length is a dimension
             sum1 += (x[i] - offset) * (x[i] - offset);
             sum2 += Math.cos(2 * Math.PI * (x[i] - offset));
         }
         return -20.0*Math.exp(-0.2*Math.sqrt(sum1 / (double) x.length))
+ 20 - (Math.exp(sum2 / (double) x.length)) + Math.exp(1);
    }
        @Override
        public double getOffset() {
              return offset;
        }
        @Override
        public double getMaxSpawnPoint() {
              return 32;
        }
        @Override
        public double getMinSpawnPoint() {
              return 16;
        }

}
```

ก.1.3 Class Griewank

```
package evaluateFunctions;
public class Griewank extends FitnessFunction
{
        private static final double MAXVALUE = 600;
        private static final double MINVALUE = -600;
       private double offset = 0;

        public Griewank() {
        }

        public Griewank(double offset) {
              this.offset = offset;
        }
        @Override
        public double getMaxSpawnPoint() {
              return 600;
        }
        @Override
        public double getMinSpawnPoint() {
              return 300;//-600;
        }

        @Override
```

```
	public double getMaxValue() {
		return MAXVALUE;
	}

	@Override
	public double getMinValue() {
		return MINVALUE;
	}

	@Override public double execute(double[] x) {
	super.execute(x);
		double sum = 0,pi = 1.0;
		for(int i = 0 ; i < x.length ; i++)
		{
			sum += ((x[i] - offset) * (x[i] - offset));
			pi *= Math.cos((x[i] - offset) / Math.sqrt(i + 1));
		}
		return sum / 4000 - pi + 1;
	}

	public double getOffset() {
		return offset;
	}


}
```

ก.1.4 Class Rastrigin

```
package evaluateFunctions;
public class Rastrigin extends FitnessFunction {
    private double offset = 0;

	public Rastrigin() {
	}

	public Rastrigin(double offset) {
		this.offset = offset;
	}

	@Override
	public double getMaxValue() {
		return 5.12;
	}
	@Override
	public double getMaxSpawnPoint() {
		return 5.12;
	}
	@Override
	public double getMinSpawnPoint() {
```

```
		return 2.56;
	}

	@Override
	public double getMinValue() {
		return -5.12;
	}

	@Override
	public double execute(double[] x) {
	super.execute(x);
		double sum = 0;
		for(int i = 0 ; i < x.length ; i++)
		{
			sum += (x[i] - offset) * (x[i] - offset) - 10 *
Math.cos(2 * Math.PI * (x[i] - offset));
		}
		return 10 * x.length + sum;
	}

	public double getOffset() {
		return offset;
	}

}
```

ก.1.5 Class Sphere

```
package evaluateFunctions;
public class Sphere extends FitnessFunction{
	private double offset = 0;
	public Sphere(double offset) {
		this.offset = offset;
	}

	public Sphere()
	{

	}

	@Override
	public double getMaxValue() {
		return 300;
	}

	@Override
	public double getMinValue() {
		return -300;
	}
	@Override
	public double getMaxSpawnPoint() {
```

```
		return 300;
	}
	@Override
	public double getMinSpawnPoint() {
		return 150;
	}

	@Override public double execute(double[] x) {
	super.execute(x);
		double sum = 0;
		for(int i = 0 ; i < x.length ; i++)
			sum += ((x[i] - offset) * (x[i] - offset));
		return sum;
	}
	public double getOffset() {
		return offset;
	}
}
```

ก.1.6 Class Rosenbrock

```
package evaluateFunctions;

public class Rosenbrock extends FitnessFunction {
	private double offset = 0;

	public Rosenbrock(double offset) {
		this.offset = offset;
	}

	public Rosenbrock() {
		// TODO Auto-generated constructor stub
	}

	@Override
	public double getMaxValue() {
		// KUY
		return 30.0;
	}

	@Override
	public double getMinValue() {
		return -30.0;
	}

	@Override
	public double execute(double[] x) {
	super.execute(x);
		double sum = 0;
		for(int i = 0 ; i < x.length - 1 ; i++)
		{
```

```
                    sum += 100 * Math.pow(((x[i + 1] - offset) -
Math.pow((x[i] - offset), 2)), 2) + Math.pow((x[i] - offset) - 1, 2);
                }
                return sum;
        }

        public double getOffset() {
                return offset;
        }

        @Override
        public double getMaxSpawnPoint() {
                return 30;
        }

        @Override
        public double getMinSpawnPoint() {
                return 15;
        }

}
```

ก.1.7 Class Schwefel

```
package evaluateFunctions;
public class Schwefel extends FitnessFunction {
        private double offset = 0;
      public Schwefel() {
                // TODO Auto-generated constructor stub
        }

        public Schwefel(double offset) {
                this.offset = offset;
        }

        @Override
      public double getMaxValue() {
          return 500;
      }

      @Override
      public double getMinValue() {
          return -500;
      }
      @Override
         public double getMaxSpawnPoint() {
                return -250;
         }
         @Override
         public double getMinSpawnPoint() {
                return getMinValue();
```

```
        }

    @Override
    public double execute(double[] x) {
       super.execute(x);
        double multiply = 418.9829 * x.length;
        double sum = 0;
        for(int i = 0 ; i < x.length ; i++)
        {
            sum += (x[i] - offset) * Math.sin(Math.sqrt(Math.abs((x[i] -
offset))));
        }
        return multiply - sum;
    }

       public double getOffset() {
             return offset;
       }
}
```

ก.1.8 Class Beale

```
package evaluateFunctions;

public class Beale extends FitnessFunction {
    public static final double OPTIMAL_VALUE = 0;
    private double offset = 0;

       public Beale() {

       }

       public Beale(double offset) {
             super();
             this.offset = offset;
       }
       @Override
       public double getMaxSpawnPoint() {
             return 4.5;
       }
       @Override
       public double getMinSpawnPoint() {
             return 2.25;
       }
       @Override
       public double getMaxValue() {
             return 4.5;
       }

       @Override
       public double getMinValue() {
```

```
		return -4.5;
	}

	@Override
	public double execute(double[] x) {
	super.execute(x);
		return  Math.pow(1.5 - (x[0] - offset) + ((x[0] - offset)
* (x[1] - offset)), 2) +
				Math.pow(2.25 - (x[0] - offset) + ((x[0] -
offset) * Math.pow((x[1] - offset), 2)), 2) +
				Math.pow(2.625 - (x[0] - offset) + ((x[0] -
offset) * Math.pow((x[1] - offset), 3)), 2);
	}

	@Override
	public double getOffset() {
		return offset;
	}

}
```

ก.1.9 Class Brown

```
package evaluateFunctions;
public class Brown extends FitnessFunction {
	private double offset = 0;

	@Override
	public double getMaxValue() {
		return 4;
	}

	@Override
	public double getMinValue() {
		return -1;
	}
	@Override
	public double getMaxSpawnPoint() {
		return getMaxValue();
	}
	@Override
	public double getMinSpawnPoint() {
		return getMinValue();
	}
	@Override
	public double execute(double[] x) {
	super.execute(x);
		double sum = 0 ;
		for(int i = 0 ; i < x.length - 1 ; i++)
		{
```

```
                        sum += Math.pow(Math.pow((x[i] - offset),
2),Math.pow((x[i + 1] - offset), 2) + 1) + Math.pow(Math.pow((x[i + 1] -
offset), 2),Math.pow((x[i] - offset), 2) + 1);
                }
                return sum;
        }

        @Override
        public double getOffset() {
                return offset;
        }

        public Brown(double offset) {
                this.offset = offset;
        }

        public Brown() {

        }

}
```

ก.1.10 Class Cigar

```
package evaluateFunctions;

public class Cigar extends FitnessFunction {
        public static final double OPTIMAL_X = 0;
      public static final double OPTIMAL_VALUE = 0;
      private double offset = 0;

        @Override
        public double getMaxValue() {
                return 10;
        }

        @Override
        public double getMinValue() {
                return -10;
        }
        @Override
        public double getMaxSpawnPoint() {
                return getMaxValue();
        }
        @Override
        public double getMinSpawnPoint() {
                return getMinValue();
        }

        @Override
```

```
        public double execute(double[] x) {
        super.execute(x);
                double sum = 0;
                for(int i = 1 ; i < x.length ; i++)
                {
                        sum += (x[i] - offset) * (x[i] - offset);
                }
                return (x[0] - offset) * (x[0] - offset) + 1000000 * sum;
        }

        @Override
        public double getOffset() {
                return offset;
        }

        public Cigar() {

        }

        public Cigar(double offset) {
                this.offset = offset;
        }

}
```

ก.1.11 Class Easom

```
package evaluateFunctions;

public class Cigar extends FitnessFunction {
        public static final double OPTIMAL_X = 0;
    public static final double OPTIMAL_VALUE = 0;
    private double offset = 0;

        @Override
        public double getMaxValue() {
                return 10;
        }

        @Override
        public double getMinValue() {
                return -10;
        }
        @Override
        public double getMaxSpawnPoint() {
                return getMaxValue();
        }
        @Override
        public double getMinSpawnPoint() {
                return getMinValue();
        }
```

```
	@Override
	public double execute(double[] x) {
	super.execute(x);
		double sum = 0;
		for(int i = 1 ; i < x.length ; i++)
		{
			sum += (x[i] - offset) * (x[i] - offset);
		}
		return (x[0] - offset) * (x[0] - offset) + 1000000 * sum;
	}

	@Override
	public double getOffset() {
		return offset;
	}

	public Cigar() {

	}

	public Cigar(double offset) {
		this.offset = offset;
	}
}
```

ก.1.12 Class Exponential

```
package evaluateFunctions;

public class Exponential extends FitnessFunction {
    public static final double OPTIMAL_X = 0;
    public static final double OPTIMAL_VALUE = 0;
    private double offset = 0;

	@Override
	public double getMaxValue() {
		return 1;
	}

	@Override
	public double getMinValue() {
		return -1;
	}

	@Override
	public double getMaxSpawnPoint() {
		return getMaxValue();
	}
	@Override
```

```
	public double getMinSpawnPoint() {
		return getMinValue();
	}
	@Override
	public double execute(double[] x) {
	super.execute(x);
		double sum = 0;
		for(int i = 0 ; i < x.length ; i++)
		{
			sum += (x[i] - offset) * (x[i] - offset);
		}
		return - Math.exp(-0.5 * sum) + 1;
	}

	@Override
	public double getOffset() {
		return offset;
	}

	public Exponential() {

	}

	public Exponential(double offset) {
		this.offset = offset;
	}

}
```

ก.1.13 Class GoldsteinPiece

```
package evaluateFunctions;

public class GoldsteinPiece extends FitnessFunction {
	public static final double OPTIMAL_X = 0; //y = -1
    public static final double OPTIMAL_VALUE = 3;
    private double offset = 0;

	@Override
	public double getMaxValue() {
		return 2;
	}
	@Override
	public double getMaxSpawnPoint() {
		return getMaxValue();
	}
	@Override
	public double getMinSpawnPoint() {
		return getMinValue();
	}
```

```
	@Override
	public double getMinValue() {
		return -2;
	}

	@Override
	public double execute(double[] x) {
	super.execute(x);
		return (1 +
				(Math.pow((x[0] - offset) + (x[1] - offset) +
1 , 2) *
				(19 - 14 * (x[0] - offset) + 3 *
Math.pow((x[0] - offset),2) - 14 * (x[1] - offset) + 6 * (x[0] - offset)
* (x[1] - offset) + 3 * Math.pow((x[1] - offset),2)))) *
				(30 +
					(Math.pow(2 * (x[0] - offset) -
3 * (x[1] - offset) , 2) *
					(18 - 32 * (x[0] - offset) + 12
* Math.pow((x[0] - offset),2) + 48 * (x[1] - offset) - 36 * (x[0] -
offset) * (x[1] - offset) + 27 * Math.pow((x[1] - offset),2))));
	}

	@Override
	public double getOffset() {
		return offset;
	}

	public GoldsteinPiece() {

	}

	public GoldsteinPiece(double offset) {
		this.offset = offset;
	}

}
```

ก.1.14 Class Holder

```
package evaluateFunctions;

public class Holder extends FitnessFunction {
	private double offset = 0;

	public Holder(double offset) {
		this.offset = offset;
	}

	public Holder() {
	}
```

```
	@Override
	public double getMaxValue() {
		return 10;
	}
	@Override
	public double getMaxSpawnPoint() {
		return getMaxValue();
	}
	@Override
	public double getMinSpawnPoint() {
		return getMinValue();
	}

	@Override
	public double getMinValue() {
		return -10;
	}

	@Override
	public double execute(double[] x) {
	super.execute(x);
		double exponent = Math.abs(1 - Math.sqrt(
				(Math.pow((x[0] - offset), 2) +
Math.pow((x[1] - offset), 2))
				) / Math.PI);
		return - Math.abs(Math.sin((x[0] - offset)) *
Math.cos((x[1] - offset)) * Math.exp(exponent));
	}

	public double getOffset() {
		return offset;
	}

}
```

ก.1.15 Class Levy

```
package evaluateFunctions;

public class Levy extends FitnessFunction {
	private double offset = 0;

	public Levy() {
		// TODO Auto-generated constructor stub
	}

	public Levy(double offset) {
		this.offset = offset;
	}

	@Override
```

```
	public double getMaxValue() {
		return 10;
	}

	@Override
	public double getMaxSpawnPoint() {
		return 10;
	}
	@Override
	public double getMinSpawnPoint() {
		return 5;
	}

	@Override
	public double getMinValue() {
		return -10;
	}

	@Override
	public double execute(double[] x) {
	super.execute(x);
		double sum = 0;
		for(int i = 0 ; i < x.length ; i++)
		{
			sum += Math.pow(w(x,i) - 1,2) * (1 + 10 *
Math.pow(Math.sin(Math.PI * w(x,i) + 1),2)) +
					Math.pow(w(x,x.length - 1) - 1,2) * (1
+ Math.pow(Math.sin(2 * Math.PI * w(x,x.length - 1)),2));
		}
		return Math.pow(Math.sin(Math.PI * w(x,0)),2) + sum;
	}
	private double w(double[] x,int index)
	{
		return 1 + (x[index] - 1 - offset) / 4;
	}

	public double getOffset() {
		return offset;
	}

}
```

ก.1.16 Class Multimod

```
package evaluateFunctions;

public class Multimod extends FitnessFunction {
	private double offset = 0;

	public Multimod() {
		// TODO Auto-generated constructor stub
```

```
	}

	public Multimod(double offset) {
		this.offset = offset;
	}

	@Override
	public double getMaxValue() {
		return 10;
	}

	@Override
	public double getMinValue() {
		return -10;
	}
	@Override
	public double getMaxSpawnPoint() {
		return 10;
	}
	@Override
	public double getMinSpawnPoint() {
		return 5;
	}

	@Override
	public double execute(double[] x) {
	super.execute(x);
		double sum = 0,product = 1;
		for(int i = 0 ; i < x.length - 1 ; i++)
		{
			sum += Math.abs((x[i] - offset));
			product *= Math.abs((x[i] - offset));
		}
		return sum * product;
	}

	public double getOffset() {
		return offset;
	}

}
```

ก.1.17 Class ParallelEllipsoid

```
package evaluateFunctions;

public class ParallelEllipsoid extends FitnessFunction {
	public static final double OPTIMAL_X = 0;
    public static final double OPTIMAL_VALUE = 0;
    private double offset = 0;
```

```
	public ParallelEllipsoid() {
		// TODO Auto-generated constructor stub
	}

	public ParallelEllipsoid(double offset) {
		this.offset = offset;
	}

	@Override
	public double getMaxValue() {
		return 5.12;
	}

	@Override
	public double getMinValue() {
		return -5.12;
	}
	@Override
	public double getMaxSpawnPoint() {
		return getMaxValue();
	}
	@Override
	public double getMinSpawnPoint() {
		return getMinValue();
	}

	@Override
	public double execute(double[] x) {
	super.execute(x);
		double sum = 0;
		for(int i = 1 ; i <= x.length ; i++)
			sum += i * (x[i - 1] - offset) * (x[i - 1] -
offset);
		return sum;
	}

	public double getOffset() {
		return offset;
	}

}
```

ก.1.18 Class RotatedEllipsoid

```
package evaluateFunctions;

public class RotatedEllipsoid extends FitnessFunction {
	public static final double OPTIMAL_X = 0;
    public static final double OPTIMAL_VALUE = 0;
    private double offset = 0;
```

```
	@Override
	public double getMaxValue() {
		return 65.536;
	}

	@Override
	public double getMinValue() {
		return -65.536;
	}
	@Override
	public double getMaxSpawnPoint() {
		return getMaxValue();
	}
	@Override
	public double getMinSpawnPoint() {
		return getMinValue();
	}

	@Override
	public double execute(double[] x) {
	super.execute(x);
		double sum = 0;
		for(int i = 0 ; i < x.length ; i++)
			for(int j = 0 ; j < i ; j++)
				sum += Math.pow(x[i] - offset, 2);
		return sum;
	}

	public double getOffset() {
		return offset;
	}

	public RotatedEllipsoid() {
		// TODO Auto-generated constructor stub
	}

	public RotatedEllipsoid(double offset) {
		this.offset = offset;
	}

}
```

ก.1.19 Class Schaffer

```
package evaluateFunctions;


public class Schaffer extends FitnessFunction {
    public static final double OPTIMAL_X = 0;
    public static final double OPTIMAL_VALUE = 0;
    private double offset = 0;
```

```
	public Schaffer(double offset) {
		this.offset = offset;
	}

	public Schaffer() {
		// TODO Auto-generated constructor stub
	}

	@Override
	public double getMaxValue() {
		return 100;
	}
	@Override
	public double getMaxSpawnPoint() {
		return 100;
	}
	@Override
	public double getMinSpawnPoint() {
		return 50;
	}

	@Override
	public double getMinValue() {
		return -100;
	}

	@Override
	public double execute(double[] x) {
	super.execute(x);
		double top = Math.pow(Math.sin(Math.sqrt(Math.pow((x[0] -
offset), 2) + Math.pow((x[1] - offset), 2))), 2) - 0.5;
		double bottom = Math.pow(1.0 + 0.001 * (Math.pow((x[0] -
offset), 2) + Math.pow((x[1] - offset), 2)), 2);
		return 0.5 + top / bottom;
	}

	public double getOffset() {
		return offset;
	}

}
```

ก.1.20 Class Shubert

```
package evaluateFunctions;

public class Shubert extends FitnessFunction {
	private double offset = 0;
    public Shubert() {
		// TODO Auto-generated constructor stub
```

```
	}

	public Shubert(double offset) {
		this.offset = offset;
	}

	public static final double OPTIMAL_VALUE = 186.7369;

	@Override
	public double getMaxValue() {
		return 10;
	}

	@Override
	public double getMinValue() {
		return -10;
	}
	@Override
	public double getMaxSpawnPoint() {
		return 10;
	}
	@Override
	public double getMinSpawnPoint() {
		return 5;
	}

	@Override
	public double execute(double[] x) {
	super.execute(x);
		double sum1 = 0,sum2 = 0;
		for(int i = 1 ; i <= 5 ; i++)
		{
			sum1 += i * Math.cos((i + 1) * (x[0] - offset) + i);
			sum2 += i * Math.cos((i + 1) * (x[1] - offset) + i);
		}
		return sum1 * sum2;
	}

	public double getOffset() {
		return offset;
	}

}
```

ก.1.21 Class Step

```
package evaluateFunctions;
public class Step extends FitnessFunction {
    public static final double OPTIMAL_X = 0.5;
    public static final double OPTIMAL_VALUE = 0;
    private double offset = 0;
```

```
	public Step(double offset) {
		this.offset = offset;
	}

	public Step() {
		// TODO Auto-generated constructor stub
	}

	@Override
	public double getMaxValue() {
	    return 100;
	}

	@Override
	public double getMinValue() {
	    return -100;
	}
	@Override
	public double getMaxSpawnPoint() {
		return getMaxValue();
	}
	@Override
	public double getMinSpawnPoint() {
		return getMinValue();
	}

	@Override
	public double execute(double[] x) {
	    super.execute(x);
	    double sum = 0;
	    for(int i = 0 ; i < x.length ; i++)
	    {
	        sum += Math.pow((Math.floor(x[i] - offset) + 0.5),2);
	    }
	    return sum;
	}

	public double getOffset() {
		return offset;
	}
}
```

ก.1.22 Class Trid

```
package evaluateFunctions;

public class Trid extends FitnessFunction {
	private double offset = 0;
	private int dimension = 0;

	public Trid(double offset,int dimension) {
```

```
		this.offset = offset;
		this.dimension = dimension;
	}

	public Trid(int dimension) {
		this.dimension = dimension;
	}

	@Override
	public double getMaxValue() {
		return Math.pow(dimension, 2);
	}

	@Override
	public double getMinValue() {
		return - Math.pow(dimension, 2);
	}
	@Override
	public double getMaxSpawnPoint() {
		return getMaxValue();
	}
	@Override
	public double getMinSpawnPoint() {
		return getMinValue();
	}

	@Override
	public double execute(double[] x) {
	super.execute(x);
		double sum1 = Math.pow((x[0] - offset) - 1, 2) ,sum2 = 0;
		for(int i = 1 ; i < x.length ; i++)
		{
			sum1 += Math.pow((x[i] - offset)- 1, 2);
			sum2 += (x[i] - offset) * (x[i - 1] - offset);
		}
		return sum1 - sum2;
	}

	public double getOffset() {
		return offset;
	}

}
```

ก.1.23 Class Zakharov

```
package evaluateFunctions;
public class Zakharov extends FitnessFunction {
	private double offset = 0;
    @Override
    public double getMaxValue() {
```

```
        return 10;
    }

    @Override
    public double getMinValue() {
        return -5;
    }

    @Override
    public double execute(double[] x) {
       super.execute(x);
        double sum1 = 0,sum2 = 0;
        for(int i = 0 ; i < x.length ; i++)
        {
            sum1 += (x[i] - offset) * (x[i] - offset);
            sum2 += (i + 1) / 2 * (x[i] - offset);
        }
        return sum1 + Math.pow(sum2,2) + Math.pow(sum2,4);
    }
    @Override
       public double getMaxSpawnPoint() {
             return getMaxValue();
       }
       @Override
       public double getMinSpawnPoint() {
             return getMinValue();
       }

       public Zakharov() {
             // TODO Auto-generated constructor stub
       }

       public Zakharov(double offset) {
             this.offset = offset;
       }

       public double getOffset() {
             return offset;
       }
}
```

ก.2 Package optimizationMethod

ก.2.1 Class GenericFunctions

```
package optimizationMethods;

import dataManager.ParticleSet;
import dataManager.Particle;
import dataManager.Source;
```

```
public class GenericFunctions {
	public static Particle generateFromGBest(ParticleSet d) {// used
in GA hybrid too
		Particle p = new Particle(d.getDimension());
		p.source = Source.RANDOMIZED;
		for(int i = 0 ; i < d.getDimension() ; i++)
		{
			//p.data[i] = d.gbest.gbests[(int) (Math.random() *
d.gbest.poolSize)].data[(int) (Math.random() * d.getDimension())];
			p.data[i] = d.gbest.gbests[(int) (Math.random() *
d.gbest.numberOfPositions)].data[i];
		}
		p.answer = d.b.execute(p.data);//evaluation function
		p.updatePBest();
		return p;
	}
}
```

ก.2.2 Class GeneticsAlgorithm

```
package optimizationMethods;
import evaluateFunctions.FitnessFunction;
import dataManager.ParticleSet;
import dataManager.Particle;
import dataManager.Source;

public class GeneticsAlgorithm {

	public GeneticsAlgorithm()
	{

	}
	public void selection(ParticleSet d)
	{
		d.sort();
		d.updateGBest(d.data[0]);
		if(d.numberOfParticles > d.getSize() / 2)
d.numberOfParticles = d.getSize() / 2;
	}
	public void crossOver(ParticleSet d)
	{
		for(int i = 0 ; i < d.getSize() / 2 ; i+=2)
		{
			d.data[d.numberOfParticles] = cross(d.b, d.data[i],
d.data[i + 1]);
			d.numberOfParticles++;
		}

	}
	public Particle cross(ParticleSet d,int index1, int index2){
```

```
          //this isn't crossover, it do only cross data in every
dimension between 2 input index and return value in every dimension of
child
                    Particle child = d.data[index1].clone();
                    for(int i = 0 ; i < d.getDimension() ; i++)
                    {
                          double rand = Math.random();
                          child.data[i] = (d.data[index1].data[i] *
rand) + (d.data[index2].data[i] * (1 - rand));

                          child.source = Source.CROSSED;
                          if(child.data[i] > d.b.getMaxValue())
child.data[i] -= (child.data[i] - d.b.getMaxValue());
                          else if(child.data[i] < d.b.getMinValue())
child.data[i] -= (child.data[i] - d.b.getMinValue());
                    }
                    child.answer = d.b.execute(child.data);
                    child.updatePBest();
                    return child;

              }
              public Particle cross(FitnessFunction b,Particle p1,
Particle p2){
                  //this isn't crossover, it do only cross data in every
dimension between 2 input index and return value in every dimension of
child
                          Particle child = p1.clone();
                          for(int i = 0 ; i < p1.data.length ; i++)
                          {
                                double rand = Math.random();
                                child.data[i] = (p1.data[i] * rand) +
(p2.data[i] * (1 - rand));
                          }
                          child.source = Source.CROSSED;
                          bounding(child,b);
                          return child;

                    }
              public void mutate(ParticleSet d)// mutation chance is
between 0,100
              {
                    //int[] chosen = rouletteWheel(d,d.getSize() / 3);
                    double roll;
                    for(int i = 0 ; i < d.getSize() / 4 ; i++)
                    {
                          int index = (int) (Math.random() *
d.numberOfParticles);
                          for(int j = 0 ; j < d.getDimension() ; j++)
                          {
                                roll = Math.random();
                                if(roll < 0.1){
```

```
                                        if (roll < 0.05)//if true
increase else decrease
                                        {
                                              d.data[index].data[j] +=
(d.data[index].data[j] * 0.05 *Math.random());
                                        }
                                        else d.data[index].data[j] -=
(d.data[index].data[j] * 0.05 *Math.random());
                                    }

                              }
                              d.data[index].source = Source.MUTATED;
                              bounding(d.data[index],d.b);
                        }
                  }
            protected void bounding(Particle p,FitnessFunction function)
            {
                  for(int i = 0 ; i < p.data.length ; i++)
                  {
                        if(p.data[i] > function.getMaxValue()) p.data[i] =
function.getMaxValue() - (p.data[i] - function.getMaxValue());
                        else if(p.data[i] < function.getMinValue())
p.data[i] = function.getMinValue() - (p.data[i] -
function.getMinValue());
                  }
                  p.answer = function.execute(p.data);//evaluation function
                  p.updatePBest();
            }
      }
```

ก.2.3 Class ParticleSwarmOptimization

```
package optimizationMethods;
import evaluateFunctions.FitnessFunction;
import dataManager.ParticleSet;
import dataManager.Particle;
import dataManager.Source;

public class ParticleSwarmOptimization {

      protected final double C1 = 0.729;
      protected final double C2 = 1.496180;
      protected final double C3 = 1.496180;
      public static int out = 0;
      public ParticleSwarmOptimization() {

      }

      public Particle move(ParticleSet d, int index,double vLimit) {
            Particle result = d.data[index].clone();

            for (int j = 0; j < d.getDimension(); j++) {
```

```
                    result.v[j] = C1 * result.v[j] + C2 * Math.random()
* (result.pbest[j] - result.data[j]) + C3 * Math.random() *
(d.gbest.gbests[0].data[j] - result.data[j]);
                    if(result.v[j] > vLimit) result.v[j] = vLimit;
                    else if(result.v[j] < - vLimit) result.v[j] = -
vLimit;
                    result.data[j] = (result.data[j] + result.v[j]);//
change position
                    result.source = Source.MOVED;
              }

              boundHandle(result,d.b);
              return result;
       }

       public void move(ParticleSet d) {
              int bestIndex = 0;
              double bestEvaluate = 9e307;
              double vLimit = (d.b.getMaxValue() - d.b.getMinValue()) /
2;
              for (int i = 0; i < d.numberOfParticles; i++) {
                    d.data[i] = move(d, i,vLimit);// change position
                    if(bestEvaluate > d.data[i].answer)
                    {
                          bestEvaluate = d.data[i].answer;
                          bestIndex = i;
                    }
              }
              d.updateGBest(d.data[bestIndex]);
       }
       protected void boundHandle(Particle p,FitnessFunction function)
       {
              for(int i = 0 ; i < p.data.length ; i++)
              {
                    if(p.data[i] < function.getMinValue() || p.data[i] >
function.getMaxValue()) {
                          out++;
                          return;
                    }
              }
              p.answer = function.execute(p.data);//evaluation function
              p.updatePBest();
       }

}
```

ก.2.4 Class PSOWithHistoricalBests

```
package optimizationMethods;
```

```
import dataManager.ParticleSet;
import dataManager.Particle;

public class PSOWithHistoricalBests extends ParticleSwarmOptimization {
	public void move(ParticleSet d) {
		double vLimit = (d.b.getMaxValue() - d.b.getMinValue()) /
2;
		for (int i = 0; i < d.numberOfParticles; i++) {
			d.data[i] = move(d, i ,vLimit);// change position
			d.updateHistoricalBests(d.data[i]); //decide should
it be in historical bests or not
		}
	}

	public void generateNewFromGBest(ParticleSet d) {
		for(int i = 0 ; i < 1 ; i++)
		{
			Particle p =
GenericFunctions.generateFromGBest(d);// harmony search
			d.updateHistoricalBests(p); //decide should it be in
historical bests or not
		}
	}

}
```

ก.3 Package dataManager

ก.3.1 Class Particle

```
package dataManager;
import evaluateFunctions.FitnessFunction;
public class Particle {
    public double[] data;
    public double answer;
    public double[] v;
    public double[] pbest;
    public double pbestAnswer;
    public Source source;
    public Particle(int dimension)
    {
        data = new double[dimension];
        pbestAnswer = 9e307;
        v = new double[dimension];
    }
    public void initialize(FitnessFunction b)
    {
        for(int i = 0 ; i < data.length ; i++)
        {
```

```
            data[i] = Math.random() * (b.getMaxSpawnPoint() -
b.getMinSpawnPoint()) + b.getMinSpawnPoint();
            v[i] = 0.1 * data[i];
        }
        source = Source.INITIATED;
        answer = b.execute(data);
        pbest = this.data.clone();
        pbestAnswer = this.answer;

    }
    public Particle clone()
    {
        Particle p = new Particle(data.length);
        p.data = this.data.clone();
        p.v = this.v.clone();
        p.pbest = this.pbest.clone();
        p.pbestAnswer = this.pbestAnswer;
        p.source = source;
        p.answer = this.answer;
        return p;
    }
    public void updatePBest()
    {
        if(pbestAnswer > answer) {
            pbest = this.data.clone();
            pbestAnswer = answer;
        }
    }
    public String toString()
    {
      String s = new String();
                for(int j = 0 ; j < data.length ; j++)
                {
                    s += (String.format("%.2f", data[j]) + ",");
                }
                s += ("Answer," + String.format("%.2f", answer) +
",");
                s += (source + ",");
            return s;
    }

}
```

ก.3.2 Class ParticleSet

```
package dataManager;
import evaluateFunctions.FitnessFunction;
public class ParticleSet {
      public FitnessFunction b;
      public int numberOfParticles;
      public Particle[] data;
```

```
	private int dimension;                    // dimension of
equation function
	private int size;
	public HistoricalBestPositions gbest;

	public int getDimension() {
		return dimension;
	}
	public int getSize() {
		return size;
	}
	public ParticleSet(FitnessFunction b, int dimension, int size) {
		this.b = b;
		this.dimension = dimension;
		this.size = size;
		numberOfParticles = size / 2;
		data = new Particle[size];
		gbest = new HistoricalBestPositions();
		firstGenerate();
	}
	public void firstGenerate()
	{

		for(int i = 0 ; i < numberOfParticles ; i++)
		{
			data[i] = new Particle(dimension);
			data[i].initialize(b);
			updateGBest(data[i]);
		}

	}
	public void sort()
	{
		boolean sorted = false;
		while(!sorted)
		{
			sorted = true;
			for(int i = 0 ; i < numberOfParticles - 1 ; i++)
			{
				if(data[i].answer > data[i + 1].answer) {
					Particle temp = data[i];
					data[i] = data[i + 1];
					data[i + 1] = temp;

					sorted = false;
				}
			}
		}
	}
	public void updateHistoricalBests(Particle p)
	{
```

```
                if((p.answer < gbest.gbests[gbest.numberOfPositions -
    1].answer))
                {
                        gbest.add(p.clone());
                }
        }
        public void updateGBest(Particle p)
        {
                if(gbest.numberOfPositions == 0 || p.answer <
    gbest.gbests[0].answer)
                {
                        gbest.gbests[0] = p.clone();
                        gbest.numberOfPositions = 1;
                }
        }

        public String toString()
        {
                String s = gbest.toString();
                for(int i = 0 ; i < numberOfParticles ; i++)
                {
                        s += ("Data " + (i + 1) + ",");
                        s += data[i].toString() + "\n";

                }
                return s;
        }
        public void addParticle(Particle p)
        {
                data[numberOfParticles++] = p;
        }
}
```

ก.3.3 Enum Source

```
package dataManager;

public enum Source {
        MUTATED,INITIATED,CROSSED,MOVED,RANDOMIZED;
}
```

ก.3.4 Class HistoricalBestPositions

```
package dataManager;
public class HistoricalBestPositions {
        public Particle[] gbests;
        public int numberOfPositions;
        public static final int MAX_SIZE = 50;
        public HistoricalBestPositions()
        {
                gbests = new Particle[MAX_SIZE];
                numberOfPositions = 0;
        }
        public void add(Particle p)
```

```
	{
		if(gbests[0] == null) {
			gbests[0] = p;
			numberOfPositions++;
			return;
		}
		for(int i = 0 ; i < numberOfPositions ; i++)
		{

			if(p.answer < gbests[i].answer)
			{
				if(! isFull())
				{
					for(int j = numberOfPositions ; j > i ;
j--)
					{
						gbests[j] = gbests[j - 1];
					}

					numberOfPositions++;
				}
				else
				{
					for(int j = numberOfPositions - 1 ; j >
i ; j--)
					{
						gbests[j] = gbests[j - 1];
					}
				}
				gbests[i] = p;
				break;
			}
		}

	}
	public boolean isFull()
	{
		return numberOfPositions == MAX_SIZE;
	}

	public String toString()
	{
		String s = "";
		for(int i = 0 ; i < numberOfPositions ; i++)
		{
			s += ("GBest " + (i + 1) + ",");
			for(int j = 0 ; j < gbests[i].data.length ; j++)
			{
				s += (String.format("%.2f",
gbests[i].data[j]) + ",");
			}
```

```
                    s += ("Answer," + String.format("%.2f",
gbests[i].answer) + "," + gbests[i].source + "\n");

            }
            return s;
        }
```

ก.4 Default Package

ก.4.1 Class Main

```
import java.util.Scanner;

import evaluateFunctions.*;
import dataManager.ParticleSet;
import optimizationMethods.*;

public class Main {
        private final static int SIZE = 100; // population size * 2
        public final static int DIMENSION = 30;
        private final static int END_CALL_COUNT = 300000;
        private static double lastGBestAnswer = 9e307;
        private static int unchangedStreak = 0;
        private final static int ROUND = 1;
        private static ParticleSet d;

        public static void main(String[] args) {


            int i; // Rounds
            for (i = 0; i < ROUND ; i++) {
                FitnessFunction b;
                int j = new Scanner(System.in).nextInt();// Evaluate
Fx
                switch (j) {
                case 1:
                        b = new Ackley();
                        break;
                case 2:
                        b = new Griewank();
                        break;
                case 3:
                        b = new Rastrigin();
                        break;
                case 4:
                        b = new Sphere();
                        break;
                case 5:
                        b = new Rosenbrock();
                        break;
```

```
                default:
                        b = new Schwefel();
                        break;
                }
                b.callCount = 0;
                d = new ParticleSet(b, DIMENSION, SIZE);
                lastGBestAnswer = d.gbest.gbests[0].answer;
                unchangedStreak = 0;

                int k = new Scanner(System.in).nextInt(); // Method
                System.out.println(k + " " + j + " " + i);

                switch (k) {
                        case 1: { //Standard PSO
                                ParticleSwarmOptimization pso = new
ParticleSwarmOptimization();
                                while (!isStopConditionMeet(d)) {
                                        pso.move(d);
                                }
                                break;
                        }
                        case 2: { //GA
                                GeneticsAlgorithm g = new
GeneticsAlgorithm();
                                while (!isStopConditionMeet(d)) {

                                        g.crossOver(d);
                                        g.mutate(d);
                                        g.selection(d);

                                }
                                break;
                        }
                        case 3: { //PSO with Historical bests
                                PSOWithHistoricalBests pso = new
PSOWithHistoricalBests();
                                while (!isStopConditionMeet(d)) {
                                        pso.move(d);
                                        if(d.gbest.isFull()) {

        pso.generateNewFromGBest(d);
                                        }

                                }
                                break;
                        }
                }
                System.out.println(d.toString());
        }

}
```

```
	private static boolean isStopConditionMeet(ParticleSet d) //check
if it meet stop condition
	{
		if(d.gbest.gbests[0].answer < lastGBestAnswer){
			lastGBestAnswer = d.gbest.gbests[0].answer;
			if(unchangedStreak > 10) {
				System.out.println("Evaluation call count = "
+ d.b.callCount);
				System.out.println(d.toString());
				new Scanner(System.in).nextLine();
			}
			unchangedStreak = 0;
		}
		else unchangedStreak++;
		return d.b.callCount >= END_CALL_COUNT; //stop condition
	}

}
```