เทคนิคของเมทริกซ์มากเลขศูนย์สำหรับการจำลองวงจรไฟฟ้าในโปรแกรม “เล็ก”

นายนภดล จิตต์จรัส

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชาวิศวกรรมไฟฟ้า ภาควิชาวิศวกรรมไฟฟ้า
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2543

ISBN 974-346-391-7

A SPARSE MATRIX TECHNIQUES FOR CIRCUIT SIMULATION IN THE "LEK" PROGRAM

Mr. Noppadol Chitcharus

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Engineering in Electrical Engineering
Department of Electrical Engineering
Faculty of Engineering
Chulalongkorn University
Academic Year 2000

ISBN 974-346-391-7

หัวข้อวิทยานิพนธ์	เทคนิคของเมทริกซ์มากเลขศูนย์สำหรับการจำลองวงจรไฟฟ้าในโปรแกรม “เล็ก”
โดย	นายนภดล จิตต์จรัส
ภาควิชา	วิศวกรรมไฟฟ้า
อาจารย์ที่ปรึกษา	รองศาสตราจารย์ ดร. เอกชัย ลีลารัศมี

คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย อนุมัติให้นับวิทยานิพนธ์ฉบับนี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญามหาบัณฑิต

………………………………………….. คณบดีคณะวิศวกรรมศาสตร์
(ศาสตราจารย์ ดร. สมศักดิ์ ปัญญาแก้ว)

คณะกรรมการสอบวิทยานิพนธ์

…………………………………………… ประธานกรรมการ
(ผู้ช่วยศาสตราจารย์ ดร. วัชรพงษ์ โขวิฑูรกิจ)

…………………………………………... อาจารย์ที่ปรึกษา
(รองศาสตราจารย์ ดร. เอกชัย ลีลารัศมี)

……………………………………………กรรมการ
(ผู้ช่วยศาสตราจารย์ ดร. สมชาย ประสิทธิ์จูตระกูล)

นภดล จิตต์จรัส : เทคนิคของเมทริกซ์มากเลขศูนย์สำหรับการจำลองวงจรไฟฟ้าในโปรแกรม “เล็ก”. (A Sparse Matrix Techniques for Circuit Simulation in the “LEK” Program) อ. ที่ปรึกษา : รศ.ดร.เอกชัย ลีลารัศมี, 94 หน้า. ISBN 974-346-391-7.

วิทยานิพนธ์ฉบับนี้กล่าวถึงการใช้เทคนิคของเมทริกซ์มากเลขศูนย์เพื่อเพิ่มความเร็วในการทำงานของโปรแกรมจำลองวงจร "เล็ก" โดยใช้ประโยชน์จากความจริงที่ว่าเมทริกซ์สัมประสิทธิ์ของสมการวงจรไฟฟ้าโดยมากจะมีสมาชิกส่วนใหญ่เป็นค่าศูนยหรือเรียกว่าเป็นเมทริกซ์มากเลขศูนย์ และโปรแกรมจำเป็นต้องแก้สมการวงจรเดิมซ้ำๆกันหลายครั้ง เทคนิคดังกล่าวประกอบด้วยการใช้โครงสร้างข้อมูลเฉพาะสำหรับเก็บค่าของเมทริกซ์มากเลขศูนย์ซึ่งช่วยลดขนาดหน่วยความจำที่เก็บเมทริกซ์, การเรียงลำดับใหม่ตามวิธีของ Markowitz เพื่อลดจำนวนสมาชิกที่ไม่ใช่ค่าศูนย์ และการข้ามการคำนวณกับค่าศูนย์ในระหว่างการแก้สมการเมทริกซ์ด้วยวิธีแยกตัวประกอบแอล-ยู การใช้เทคนิคเหล่านี้มีผลให้ความซับซ้อนของการแก้สมการวงจรไฟฟ้าลดลงอย่างชัดเจน นอกจากนี้ยังมีการใช้เทคนิคการสร้างสูตรสำเร็จการแก้สมการเมทริกซ์ในรูปของรหัสแบบแปลย่อยและรหัสคำสั่งเครื่อง โดยรหัสทั้งสองเป็นขั้นตอนแก้สมการวงจรที่ไม่มีคำสั่งวนรอบหรือคำสั่งกระโดดและอ่านค่าจากสมาชิกแต่ละตัวในเมทริกซ์โดยตรง ซึ่งช่วยให้ลดเวลาแก้สมการวงจรไฟฟ้าลงได้ถึง 10 เท่า อย่างไรก็ตาม โปรแกรมจะต้องใช้หน่วยความจำเพิ่มขึ้นสำหรับเก็บรหัสทั้งสองแบบ ซึ่งขนาดหน่วยความจำที่ใช้จะแปรตามจำนวนสมาชิกที่ไม่เป็นศูนย์ในเมทริกซ์ โดยถ้าเป็นวงจรขนาดใหญ่หน่วยความจำส่วนนี้จะมีขนาดใหญ่อย่างมาก

ภาควิชา วิศวกรรมไฟฟ้า ลายมือชื่อนิสิต ....
สาขาวิชา วิศวกรรมไฟฟ้า ลายมือชื่ออาจารย์ที่ปรึกษา ....
ปีการศึกษา 2543 ลายมือชื่ออาจารย์ที่ปรึกษาร่วม ....

4170355521 : MAJOR ELECTRICAL ENGINEERING
KEY WORD: CIRCUIT SIMULATION / LEK / SPARSE MATRIX

NOPPADOL CHITCHARUS : A SPARSE MATRIX TECHNIQUES FOR CIRCUIT SIMULATION IN THE “LEK” PROGRAM. THESIS ADVISOR : ASSO.PROF.EKACHAI LEELARASMEE, 94 pp. ISBN 974-346-391-7.

This dissertation describes the implementation of sparse matrix techniques to speed up the execution of a circuit simulation program called "LEK". Its key idea is based on the facts that the coefficient matrix of the simulated circuit equation is usually sparse, i.e. consists of a large percentage of zero elements, and that the circuit equation is to be solved many times with the same zero - nonzero matrix structure. These techniques include a sparse matrix data structuring to reduce the memory storage of a sparse matrix, a Markowitz matrix reordering to reduce the amount of nonzero matrix operations and a sparse LU factorization procedure for performing only the nonzero operations. Using these techniques, the computational complexity of the circuit equation solving subroutine has been significantly reduced. Furthermore, a technique to generate a specialized runtime code for executing the sparse LU factorization is also described in details. Two types of code, namely an interpretive code and a machine code, have been implemented. These codes contain no loop nor jump instruction and access each matrix elements directly. Using these codes, the execution of the sparse matrix equation solving can be furhter reduces up to 10 times. However, these codes also require additional memory whose amount depends on the number of nonzero matrix elements and therefore can be significant for a large circuit.

Department Electrical Engineering . Student’s signature .
Field of study Electrical Engineering . Advisor’s signature .
Academic year 2543 . Co-Advisor’s signature .

กิตติกรรมประกาศ

วิทยานิพนธ์ฉบับนี้สำเร็จลุล่วงได้ด้วยความช่วยเหลืออย่างดียิ่งจาก รศ.ดร.เอกชัย ลีลารัศมี อาจารย์ที่ปรึกษาวิทยานิพนธ์ ซึ่งท่านได้ให้คำแนะนำและข้อคิดเห็นที่มีคุณค่ากับการวิจัยนี้มาโดยตลอด และเนื่องจากการวิจัยครั้งนี้ได้รับทุนจากโครงการศิษย์ก้นกุฏิ จึงขอขอบพระคุณมา ณ ที่นี้ด้วย

ขอขอบคุณห้องปฏิบัติการวิจัยระบบเชิงเลข (DSRL) ซึ่งเป็นสถานที่ทำการวิจัยมาตลอด 2 ปีการศึกษา และขอขอบคุณท่านอาจารย์ เพื่อนพี่น้องนิสิตห้องปฏิบัติการวิจัยระบบเชิงเลขทุกท่าน ที่มีส่วนช่วยเหลือทั้งข้อคิดเห็น คำแนะนำ และกำลังใจ แก่ข้าพเจ้า

สุดท้ายนี้ ผู้วิจัยใคร่ขอกราบขอบพระคุณ บิดา-มารดา ซึ่งสนับสนุนในด้านการเงินและให้กำลังใจแก่ผู้วิจัยเสมอมาจนสำเร็จการศึกษา

นภดล จิตต์จรัส

สารบัญ

หน้า

บทที่

สารบัญ (ต่อ)

หน้า

สารบัญ (ต่อ)

หน้า

สารบัญตาราง

หน้า

สารบัญภาพ

หน้า

สารบัญภาพ (ต่อ)

สารบัญภาพ (ต่อ)

หน้า

บทที่ 1

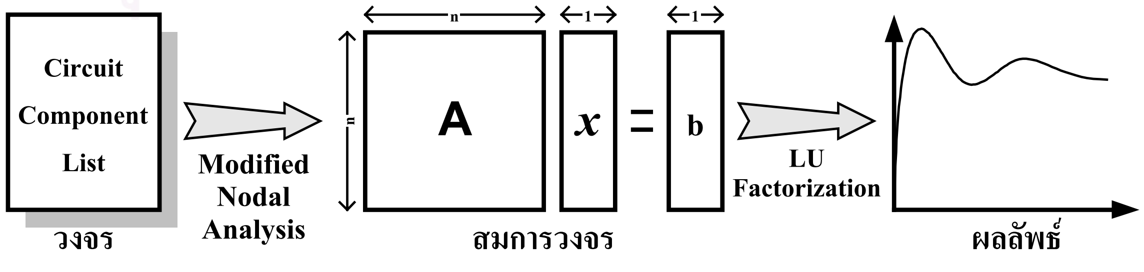
บทนำ

1.1 ความเป็นมาและความสำคัญของปัญหา

โปรแกรมจำลองการทำงานของวงจรไฟฟ้า (Circuit Simulator) เป็นโปรแกรมคอมพิวเตอร์ที่ใช้ทฤษฎีและกฎต่างๆทางไฟฟ้าเพื่อจำลองการทำงานของวงจรไฟฟ้า ช่วยให้ผู้ใช้สามารถทราบผลของวงจรไฟฟ้าที่ได้ก่อนต่อวงจรจริงหรือใช้วัดค่าตัวแปรที่ยากต่อการวัดในทางปฏิบัติ เราสามารถแบ่งประเภทการจำลองวงจรได้เป็น 2 กลุ่ม คือ กลุ่มการจำลองเชิงตัวเลข (Numerical Simulation) และกลุ่มการจำลองเชิงสัญลักษณ์ (Symbolic Simulation)

โปรแกรมในกลุ่มการจำลองเชิงสัญลักษณ์เช่น ISAAC[1] และ SNAP[2][3] มีข้อดีคือแสดงผลของการวิเคราะห์ให้ติดอยู่ในรูปของสัญลักษณ์ค่าพารามิเตอร์ (Parameter) ต่างๆในวงจร ทำให้ผู้ใช้ทำความเข้าใจถึงการทำงานของวงจรได้อย่างดี แต่โปรแกรมจำลองวงจรไฟฟ้าส่วนใหญ่จะใช้การจำลองเชิงตัวเลขเนื่องจากใช้หน่วยความจำน้อยกว่า, สามารถจำลองได้ทั้งวงจรเชิงเส้นและไม่เป็นเชิงเส้น และได้ผลการจำลองใกล้เคียงกับการทำงานจริงมากจนเป็นที่ยอมรับกันทั่วไป ตัวอย่างของโปรแกรมจำลองเชิงตัวเลขนั้น ได้แก่ SPICE[4], เล็ก[5]-[7] เป็นต้น

“เล็ก 6.0” เป็นโปรแกรมจำลองเชิงตัวเลขตัวหนึ่งที่มีประสิทธิภาพมากสำหรับใช้ในการวิเคราะห์วงจรไฟฟ้า ได้รับการพัฒนาขึ้นโดย รศ.ดร.เอกชัย ลีลารัศมี หัวหน้าห้องปฏิบัติการวิจัยระบบเชิงเลข ภาควิชาวิศวกรรมไฟฟ้า คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย เมื่อปี 2534 พัฒนาด้วยภาษาปาสคาล ใช้วิธีโมดิฟายด์โนดัล (Modified Nodal)[8] เพื่อสร้างสมการวงจรไฟฟ้าในรูปของสมการเมทริกซ์และแก้สมการด้วยวิธีแยกตัวประกอบแอล-ยู (LU-Factorization)[9] ดังรูปที่ 1.1 แต่หากเป็นการจำลองวงจรไม่เป็นเชิงเส้นจะใช้วิธี Newton Raphson Iteration[9] ช่วยในการหาคำตอบ



รูปที่ 1.1 ขั้นตอนการทำงานอย่างง่ายของโปรแกรมวิเคราะห์วงจร “เล็ก”

ที่ถูกต้อง นอกจากนี้ยังใช้วิธี Backward Euler Integration[9] สำหรับแก้สมการอนุพันธ์ (Differential Equation) ในการวิเคราห์ผลตอบสนองชั่วขณะด้วย

ในการวิเคราะห์วงจรไฟฟ้าตามวิธีโมดิฟายด์โนดัล จะเลือกค่าแรงดันไฟฟ้า ณ ปม (node) ต่างๆในวงจรไฟฟ้าเป็นตัวแปรอิสระของวงจร และสร้างสมการตามกฎกระแส (Kirchhoff's Current Law) ที่ปมทุกปมในวงจรในรูปของสมการเมทริกซ์ จะได้สมการที่เรียกว่า “Nodal Equation” ดังนี้

$$[A]_{n\times n}[x]_{n\times 1} = [b]_{n\times 1} \qquad (1.1)$$

โดย n คือ จำนวนปมทั้งหมดในวงจร โดยไม่รวมกราวด์ (ground หรือ reference node)
A คือ เมทริกซ์ค่าแอดมิตแตนซ์ (admittance) ขององค์ประกอบต่างๆที่ต่ออยู่ในวงจร
x_i คือ ค่าแรงดันไฟฟ้าของปมที่ i เมื่อเทียบกับกราวด์
b_i คือ ปริมาณกระแสไฟฟ้าจากแหล่งกระแสที่ไหลเข้าสู่ปมที่ i

สำหรับวงจรไฟฟ้าโดยทั่วไปโดยเฉพาะวงจรที่มีจำนวนปม (n) มาก มักพบว่าเมทริกซ์ A ของสมการ(1.1) จะมีสมาชิกส่วนใหญ่มีค่าเป็นศูนย์หรือเรียกได้ว่าเมทริกซ์ A เป็นเมทริกซ์มากเลขศูนย์ (Sparse Matrix) ทั้งนี้เนื่องจากค่าสมาชิกแถวที่ i คอลัมน์ที่ j ในเมทริกซ์ A (A_{ij}) จะมีค่าไม่เป็นศูนย์ก็ต่อเมื่อระหว่างปมที่ i และปมที่ j มีองค์ประกอบใดองค์ประกอบหนึ่งของวงจรต่อเชื่อมปม 2 ปมนี้กันอยู่ และสำหรับวงจรไฟฟ้าส่วนใหญ่แล้ว ปมๆหนึ่งในวงจรจะมีองค์ประกอบต่ออยู่กับปมอื่นเป็นจำนวนน้อยมากเมื่อเทียบกับจำนวนปมทั้งหมดที่มีอยู่ในวงจร ปมคู่ใดที่ไม่มีองค์ประกอบใดต่อเชื่อมกันอยู่จะทำให้ค่าสมาชิกในเมทริกซ์ A ที่ตำแหน่งนั้นมีค่าเป็นศูนย์

การที่มีสมาชิกที่มีค่าเป็นศูนย์ในเมทริกซ์ A เป็นจำนวนมาก ทำให้มีการบวก,ลบ,คูณและหารกับค่าศูนย์เป็นจำนวนมากซึ่งเป็นการคำนวณที่ไม่จำเป็น ดังนั้นหากเราสามารถลดหรือตัดการคำนวณในส่วนนี้ลงได้ จะทำให้โปรแกรมใช้เวลาในการจำลองวงจรไฟฟ้าลดลงได้มาก นอกจากนี้ยังมีเทคนิคที่ช่วยเสริมให้ใช้เวลาแก้สมการเมทริกซ์น้อยลงอีก ได้แก่

- เทคนิคการเรียงลำดับใหม่ ซึ่งเป็นการสลับตำแหน่งแถวและคอลัมน์ในเมทริกซ์ A เพื่อลดการเกิดสมาชิกที่ไม่เป็นศูนย์ในเมทริกซ์ให้น้อยลง
- เทคนิคการกลับเศษส่วนค่าสมาชิกในแนวทแยงมุขสำคัญ ซึ่งช่วยเปลี่ยนการหารให้เป็นการคูณในระหว่างการแก้สมการวงจรด้วยวิธีแยกตัวประกอบแอล-ยู
- เทคนิคการสร้างสูตรสำเร็จการแก้สมการเมทริกซ์ เป็นการสร้างการคำนวณลัดในการแก้สมการเมื่อจำเป็นต้องแก้สมการวงจรเดิมซ้ำกันหลายๆรอบ ทำให้ตัดขั้นตอนที่ไม่จำเป็นในระหว่างการแก้สมการทิ้งไปได้

อนึ่งโปรแกรม “เล็ก” เป็นโปรแกรมจำลองการทำงานของวงจรไฟฟ้าที่ดีตัวหนึ่ง แต่เนื่องจากขั้นตอนในส่วนการคำนวณผลลัพธ์ยังคงใช้กระบวนแก้สมการแบบทั่วไป จึงเหมาะสมที่จะใช้เทคนิคของเมทริกซ์มากเลขศูนย์มาประยุกต์ใช้แทนขั้นตอนเดิมเพื่อเพิ่มความเร็วในการคำนวณให้ดียิ่งขึ้น

1.2 วัตถุประสงค์ของการวิจัย

1. เพื่อศึกษาเทคนิคต่างๆในการดำเนินการกับเมทริกซ์มากเลขศูนย์ สำหรับการจำลองการทำงานของวงจรไฟฟ้า
2. เพื่อนำเทคนิคของเมทริกซ์มากเลขศูนย์มาประยุกต์ใช้แทนขั้นตอนการคำนวณผลลัพธ์แบบเดิมของโปรแกรม “เล็ก” เพื่อลดเวลาและหน่วยความจำที่ใช้ในการคำนวณ

1.3 ขอบเขตของวิทยานิพนธ์

1. ออกแบบโครงสร้างข้อมูลในการเก็บเมทริกซ์มากเลขศูนย์ ที่ประหยัดหน่วยความจำและยังรองรับขั้นตอนการแก้สมการเมทริกซ์ด้วยวิธีแยกตัวประกอบแอล-ยู
2. สามารถพัฒนาขั้นตอนการแก้สมการเมทริกซ์ด้วยวิธีแยกตัวประกอบแอล-ยูโดยใช้ประโยชน์จากโครงสร้างข้อมูลที่ออกแบบไว้ตามข้อ 1
3. สามารถแปลงรหัสดำเนินการสำหรับการแก้สมการเมทริกซ์ด้วยวิธีแยกตัวประกอบแอล-ยู ให้เป็นรหัสคำสั่งของหน่วยประมวลผล(Machine Code) เพื่อลดเวลาการคำนวณลงให้มากที่สุด
4. สามารถดัดแปลงโปรแกรม “เล็ก” ให้สามารถใช้ขั้นตอนวิธีใหม่นี้ในการจำลองวงจรไฟฟ้าทั้งการจำลองแบบวิเคราะห์หาจุดทำงานสงบ(DC Operation point), แบบผลตอบสนองชั่วขณะ(Transient Response) และแบบผลตอบสนองเชิงความถี่ (Frequency Response)

1.4 ขั้นตอนการดำเนินงาน

1. ศึกษาขั้นตอนวิธีการแก้ปัญหาของโปรแกรมจำลองวงจรไฟฟ้า และรูปแบบเฉพาะของเมทริกซ์สัมประสิทธิ์ของสมการวงจรไฟฟ้า
2. ศึกษาและค้นคว้าหาขั้นตอนวิธีการดำเนินการกับเมทริกซ์มากเลขศูนย์ เพื่อแทนขั้นตอนวิธีแบบเดิม
3. ทดลองนำขั้นตอนวิธีใหม่ที่ได้มาแทนลงไปในโปรแกรมเดิม และทดลองใช้งานหาข้อผิดพลาดแล้วปรับปรุงให้มีประสิทธิภาพมากยิ่งขึ้น

4. ทดลองจับเวลาและเนื้อที่หน่วยความจำที่ใช้เพื่อการจำลองวงจรไฟฟ้าต่างๆ เทียบกับโปรแกรมเดิม เพื่อคำนวณประสิทธิภาพที่เพิ่มขึ้นของโปรแกรม
5. เขียนวิทยานิพนธ์และรายงานสรุปผล

1.5 ประโยชน์ที่คาดว่าจะได้รับ

1. ได้เรียนรู้ถึงขั้นตอนวิธีและหลักการการทำงานของโปรแกรมจำลองวงจรไฟฟ้า “เล็ก”
2. ได้โปรแกรมจำลองวงจรไฟฟ้าตัวใหม่ที่สามารถทำการจำลองได้อย่างรวดเร็ว มีความสามารถมากยิ่งขึ้น เช่น สามารถจำลองวงจรไฟฟ้าที่มีจำนวนปมมากขึ้นหรือมีความซับซ้อนมากขึ้นได้

บทที่ 2

ทฤษฎีพื้นฐาน

การจำลองการทำงานของวงจรไฟฟ้านั้นมีทฤษฎีที่เกี่ยวข้องจำนวนมาก ดังนั้นในบทนี้จะนำเสนอทฤษฎีที่เกี่ยวข้องในการทำวิทยานิพนธ์นี้อย่างพอสังเขป โดยจะแยกอธิบายเป็น 2 หัวข้อ คือ ทฤษฎีในการวิเคราะห์วงจรไฟฟ้า และ ทฤษฎีทางด้านวิธีเชิงตัวเลข (Numerical Method) ซึ่งขั้นตอนทั้งหมดนี้เป็นวิธีจำลองเชิงตัวเลขที่ถูกเลือกใช้ในโปรแกรมจำลองการทำงานของวงจรไฟฟ้า “เล็ก 6.0” และเป็นขั้นตอนวิธีเดียวกับที่ใช้ในโปรแกรมจำลองเชิงเลขอื่นทั่วไป เช่น SPICE

2.1 ทฤษฎีในการวิเคราะห์วงจรไฟฟ้า

2.1.1 ตัวแปรของวงจร

ตัวแปรอิสระทุกตัวที่โปรแกรมจำลองวงจรไฟฟ้าใช้ในการเฉลยเพื่อหาคำตอบของวงจร เรียกว่า ตัวแปรวงจร ซึ่งมีหลายประเภท ได้แก่

- แรงดันปม (Node voltages)
- แรงดันกิ่ง (Branch voltages)
- กระแสกิ่ง (Branch currents)
- กระแสวนรอบ (Loop currents) ฯลฯ

แต่ตัวแปรวงจรแบบที่นิยมใช้กันมากที่สุดคือ แรงดันปม โดยที่แรงดันปมคือ ค่าแรงดันระหว่างปมนั้นและกราวด์ (Ground) เนื่องจากโดยทั่วไปแล้วแรงดันปมคือตัวแปรที่ผู้ใช้ต้องการทราบค่า และเมื่อทราบแรงดันปมของทุกๆปมแล้วจะสามารถคำนวณหาค่าตัวแปรอื่นๆในวงจรได้โดยง่าย

2.1.2 การสร้างสมการเมทริกซ์ของวงจรด้วยวิธีโมดิฟายด์โนดัล

ในทางทฤษฎีวงจร วงจรไฟฟ้าวงจรหนึ่งอาจมีสมการวงจรได้หลายแบบ ขึ้นอยู่กับการกำหนด ตัวแปรของวงจรและกฎทางไฟฟ้าที่ใช้ในการสร้างสมการ วิธีสร้างสมการที่ใช้กันทั่วไปมี 4 วิธีตาม ตารางที่ 2.1

ชื่อวิธี	ตัวแปรของวงจร	สมการวงจรทางไฟฟ้า
Node Analysis	Node voltage	กฎกระแส (Kirchoff current Law)
Mesh Analysis	Mesh current	กฎแรงดัน (Kirchoff voltage Law)
Loop Analysis	Loop current	กฎแรงดัน
Cut-set Analysis	Tree branch voltage	กฎกระแส

ตารางที่ 2.1 ตารางเปรียบเทียบวิธีต่างๆในการสร้างสมการวงจร

ดังที่ได้กล่าวมาแล้วว่าวิธี Node Analysis เป็นวิธีที่ใช้กันมากที่สุดแต่วิธีนี้มีจุดอ่อนตรงที่ไม่สามารถนำไปใช้โดยตรงกับวงจรบางชนิดได้ เช่นวงจรที่มีแหล่งกำเนิดแรงดันต่ออยู่ ดังตัวอย่างวงจรในรูปที่ 2.1 การสร้างสมการวงจรในรูปที่ 2.1 ด้วยวิธี Node Analysis นั้นจำเป็นต้องอาศัยการแปลงแหล่งกำเนิดแรงดันให้เป็นแหล่งกำเนิดกระแสก่อนจึงสามารถสร้างสมการได้ จากสาเหตุข้างต้นจึงได้มีผู้เสนอวิธีสร้างสมการแบบอื่นๆ โดยอาศัยการดัดแปลงจากวิธี Node Analysis และให้ชื่อวิธีใหม่นี้ว่า วิธีโมดิฟายด์โนดัล (Modified Nodal) วิธีนี้จะยอมให้กระแสของแหล่งกำเนิดแรงดันและกระแสของ ตัวเหนี่ยวนำเป็นตัวแปรของวงจรได้โดยไม่ต้องมีการเปลี่ยนแปลงวงจรเดิมเสียก่อน ทำให้สะดวกต่อการนำไปใช้สร้างโปรแกรมจำลองวงจรไฟฟ้า

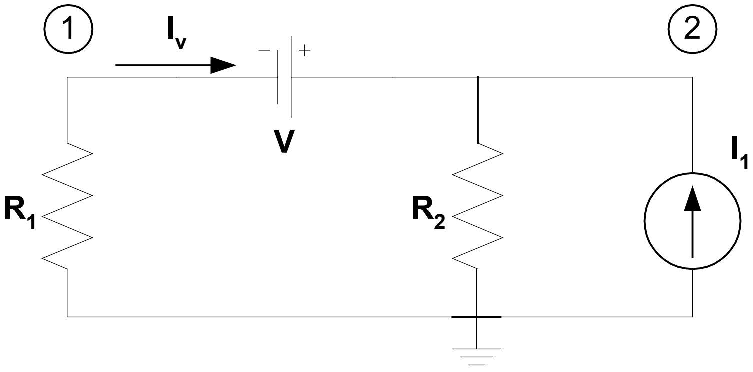
2.1.3 ตัวอย่างการสร้างสมการเมทริกซ์ของวงจรไฟฟ้า

จากวงจรในรูปที่ 2.1 ซึ่งเป็นวงจรขนาด 2 ปมที่ไม่สามารถใช้วิธี Node Analysis วิเคราะห์วงจรได้โดยตรง แต่ถ้าใช้วิธีโมดิฟายด์โนดัลในการสร้างสมการวงจรไฟฟ้าจะยอมให้ค่ากระแสผ่านแหล่งกำเนิดแรงดันเป็นตัวแปรวงจรด้วย นั้นคือจะกำหนดให้มีตัวแปรวงจร 3 ตัวคือ ค่าแรงดันที่ปม 1 (V_1), ค่าแรงดันที่ปม 2 (V_2) และค่ากระแสที่ไหลผ่านแหล่งกำเนิดแรงดัน (I_V) และสมการวงจรที่สร้างโดยวิธีโมดิฟายด์โนดัลนั้นสามารถเขียนได้ดังนี้

KCL_1 : $$\frac{V_1}{R_1} + I_V = 0 \tag{2.1}$$

KCL_2 : $$\frac{V_2}{R_2} - I_V = I_1 \tag{2.2}$$

BR_V : $$-V_1 + V_2 = V \tag{2.3}$$



รูปที่ 2.1 ตัวอย่างวงจรที่ไม่สามารถใช้วิธี Node Analysis สร้างสมการโดยตรงได้

จากตัวอย่างนี้จะพบว่าเราสามารถใช้วิธีโมดิฟายด์โนดัลในการสร้างวงจรได้ และหลังจากที่สามารถสร้างสมการวงจรได้แล้ว ต่อไปจะเป็นขั้นตอนในการแก้สมการ โดยที่เนื้อหาโดยรวมจะเป็น เนื้อหาในส่วนของทฤษฎีทางด้านเชิงตัวเลข (Numerical Method)

2.2 <u>ทฤษฎีทางด้านวิธีเชิงตัวเลข</u>

ในการแก้สมการวงจร เราสามารถเขียนให้อยู่ในรูปของสมการเมทริกซ์

$$\boldsymbol{A\,x = b} \qquad (2.4)$$

โดยที่ A เป็นเมทริกซ์ของสัมประสิทธิ์ขนาด n x n มิติ

x เป็นเวกเตอร์ของตัวแปรวงจรขนาด n x 1 มิติ

b เป็นเวกเตอร์ค่าคงที่ขนาด n x 1 มิติ

จากสมการ (2.1)-(2.3) ของวงจรในรูปที่ 2.1 ที่สร้างโดยวิธีโมดิฟายด์โนดัล นำมาเขียนใหม่ให้อยู่ในรูปของเมทริกซ์จะได้ผลลัพธ์แสดงไว้ในรูปที่ 2.2

$$\begin{bmatrix} 1/R_1 & 0 & 1 \\ 0 & 1/R_2 & -1 \\ -1 & 1 & 0 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ I_V \end{bmatrix} = \begin{bmatrix} 0 \\ I_1 \\ V \end{bmatrix}$$

รูปที่ 2.2 สมการเมทริกซ์ของวงจรในรูปที่ 2.1

หลังจากที่จัดสมการวงจรให้อยู่ในรูปสมการเมทริกซ์ $Ax = b$ ได้แล้วนั้น ต่อไปจะเป็นขั้นตอนในการแก้สมการเมทริกซ์ ซึ่งมีมากมายหลายวิธี เช่น วิธี Gaussian Elimination ซึ่งเป็นวิธีที่รู้จักกันแพร่หลายและเป็นที่นิยมใช้กันทั่วไป แต่วิธีที่โปรแกรมจำลองการทำงานของวงจรไฟฟ้านิยมเลือกใช้คือ วิธีการแยกตัวประกอบแอล-ยู (LU-Factorization)

2.2.1 การแก้สมการเมทริกซ์ด้วยวิธีแยกตัวประกอบแอล-ยู

วิธีการแยกตัวประกอบแบบแอล-ยู จะแบ่งขั้นตอนในการแก้ปัญหาออกเป็น 2 ขั้นตอนดังนี้

2.2.1.1 *Factorization*

เป็นการคำนวณเพื่อที่จะแยกเมทริกซ์ A ให้เป็น

$$\boldsymbol{A} = \boldsymbol{L}\,\boldsymbol{U} \tag{2.5}$$

โดยที่ L คือ เมทริกซ์สามเหลี่ยมข้างล่าง (Lower triangular matrix)

U คือ เมทริกซ์สามเหลี่ยมข้างบน (Upper triangular matrix) ที่มีค่าสมาชิกในแนวทแยงมุขสำคัญเป็น 1 ทั้งหมด

$$L = \begin{bmatrix} l_{11} & & & & \\ l_{21} & l_{22} & & 0 & \\ l_{31} & l_{32} & l_{33} & & \\ \dots & \dots & \dots & \dots & \\ l_{n1} & l_{n2} & l_{n3} & \dots & l_{nn} \end{bmatrix} \qquad U = \begin{bmatrix} 1 & u_{12} & u_{13} & \dots & u_{1\mathrm{n}} \\ & 1 & u_{23} & \dots & u_{2\mathrm{n}} \\ & & 1 & \dots & u_{3\mathrm{n}} \\ & 0 & & 1 & \vdots \\ & & & & 1 \end{bmatrix}$$

รูปที่ 2.3 รูปแบบของเมทริกซ์ L และ U

ค่าในเมทริกซ์ L และ U สามารถคำนวณได้จากโดยวิธี Crout method[9] ตามสูตร

$$l_{ik} = a_{ik} - \sum_{m=1}^{k-1} l_{im} u_{mk} \qquad \text{โดย } i \geq k \tag{2.6}$$

$$u_{kj} = \left(a_{kj} - \sum_{m=1}^{k-1} l_{km} u_{mj} \right) \Big/ l_{kk} \qquad \text{โดย } j > k \tag{2.7}$$

เพื่อเป็นการประหยัดหน่วยความจำ โปรแกรมจำลองวงจรไฟฟ้าจะเก็บค่าของเมทริกซ์ L และ U ไว้ในเมทริกซ์เดียวกัน โดยค่าของเมทริกซ์ L จะอยู่ที่ส่วนสามเหลี่ยมล่าง และค่าของเมทริกซ์ U จะอยู่ที่ส่วนสามเหลี่ยมบน ดังตัวอย่างของเมทริกซ์ขนาด 4 x 4 ที่แสดงในรูปที่ 2.4

$$L:U = L+U-I = \begin{bmatrix} l_{11} & u_{12} & u_{13} & u_{14} \\ l_{21} & l_{22} & u_{23} & u_{24} \\ l_{31} & l_{32} & l_{33} & u_{34} \\ l_{41} & l_{42} & l_{43} & l_{44} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11} & a_{12}/l_{11} & a_{13}/l_{11} & a_{14}/l_{11} \\ a_{21} & a_{22}-l_{21}u_{12} & (a_{23}-l_{21}u_{13})/l_{22} & (a_{24}-l_{21}u_{14})/l_{22} \\ a_{31} & a_{32}-l_{31}u_{12} & a_{33}-l_{31}u_{13}-l_{32}u_{23} & (a_{34}-l_{31}u_{14}-l_{32}u_{24})/l_{33} \\ a_{41} & a_{42}-l_{41}u_{12} & a_{43}-l_{41}u_{13}-l_{42}u_{23} & a_{44}-l_{41}u_{14}-l_{42}u_{24}-l_{43}u_{34} \end{bmatrix}$$

รูปที่ 2.4 ตัวอย่างการเก็บเมทริกซ์ L และ U ไว้ในเมทริกซ์เดียวกัน

2.2.1.2 *Substitution*

เมื่อทำ Factorization ตามสมการ (2.6) และ (2.7) แล้วจะได้สมการ

$$\boldsymbol{LU\,x = b} \tag{2.8}$$

จากสมการ (2.8) การแก้สมการเพื่อหาค่าเวกเตอร์ x ยังสามารถแบ่งการคำนวณออกเป็น 2 ขั้นตอนย่อยด้วยกัน คือ

- *Forward elimination*

ถ้าเรากำหนดให้ $U\,x = z$ จะได้ว่า $L\,z = b$ ซึ่งสามารถคำนวณค่า Z ได้โดยง่ายด้วย สมการทั่วไปได้คือ

$$z_1 = b_1/l_{11}$$

$$z_i = \left(b_i - \sum_{j=1}^{i-1} l_{ij} z_j \right) \Big/ l_{ii} \qquad \text{โดย } i = 2, 3, 4, \ldots, n \tag{2.9}$$

- *Backward elimination*

เมื่อคำนวณค่า Z ได้แล้ว จะคำนวณหาค่าตัวแปรวงจร x ได้ด้วยการแก้สมการ $U\,x = z$ ซึ่งจะได้ว่า

$$x_n = z_n$$

$$x_i = z_i - \sum_{j=i+1}^{n} u_{ij} z_j \qquad \text{โดย } i = n-1, n-2, \ldots, 1 \tag{2.10}$$

เมื่อได้คำตอบของค่าตัวแปรวงจร x แล้วจึงนำผลลัพธ์มาแสดงผลเป็นขั้นตอนสุดท้าย

บทที่ 3

โปรแกรมจำลองการทำงานของวงจรไฟฟ้า “เล็ก 6.0”

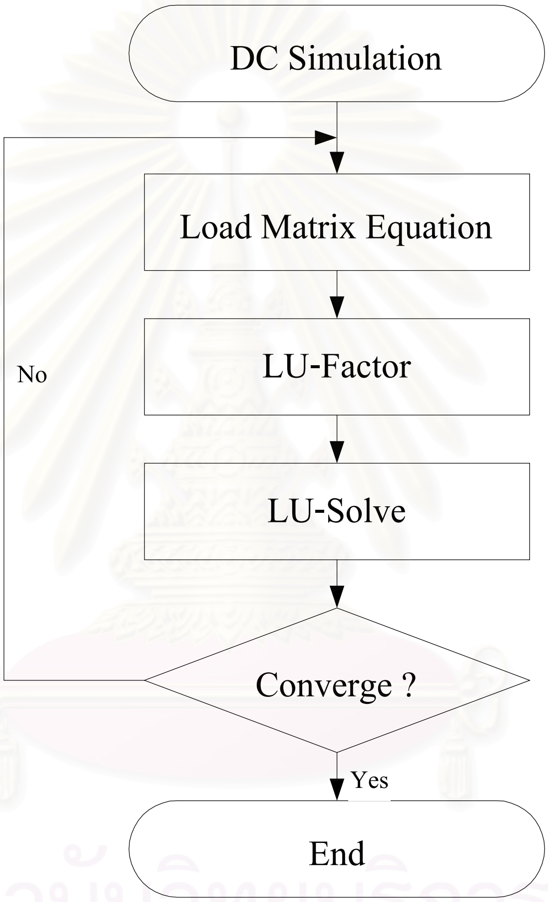
การวิจัยครั้งนี้ได้เลือกโปรแกรมจำลองการทำงานของวงจรไฟฟ้า “เล็ก 6.0” เป็นตัวแทนของโปรแกรมจำลองวงจรเชิงตัวเลขเพื่อนำมาพัฒนาเพิ่มเติมเทคนิคของเมทริกซ์มากเลขศูนย์เข้าไป ดังนั้นบทนี้จึงอธิบายขั้นตอนการการทำงานของโปรแกรม “เล็ก 6.0” ที่ใช้ในการจำลองวงจรไฟฟ้า ซึ่งหลักการทำงานของขั้นตอนวิธีจะเป็นแบบเดียวกับโปรแกรมจำลองการทำงานเชิงตัวเลขอื่นๆ ทั่วไป

“เล็ก 6.0” เป็นโปรแกรมจำลองการทำงานของวงจรไฟฟ้าที่มีขนาดเล็กมาก คือ ประมาณ 100 Kbytes เท่านั้น แต่มีความสามารถจำลองการทำงานของวงจรไฟฟ้าได้ถึง 3 โมดการทำงาน คือ

1. วิเคราะห์หาจุดทำงานสงบของวงจร (DC Simulation)
2. วิเคราะห์ผลตอบสนองเชิงความถี่ (Frequency Simulation)
3. วิเคราะห์ผลตอบสนองชั่วขณะ (Transient Simulation)

3.1 การวิเคราะห์หาจุดทำงานสงบ

ในการวิเคราะห์หาจุดทำงานสงบของวงจร โปรแกรมจะสร้างสมการเมทริกซ์ของวงจรไฟฟ้าจากข้อมูลที่ผู้ใช้ป้อนเข้ามา โดยที่ค่าคำตอบของวงจรทั้งหมดนั้นจะถูกจัดอยู่ในเมทริกซ์ตัวแปร x ของสมการเมทริกซ์ จากนั้นจึงแก้สมการเมทริกซ์ด้วยวิธีการแยกตัวประกอบแอล-ยูจึงจะได้ค่าผลลัพธ์ซึ่งเป็นค่าของตัวแปรอิสระเพียงชุดคำตอบเดียวเพื่อแสดงออกทางหน้าจอ



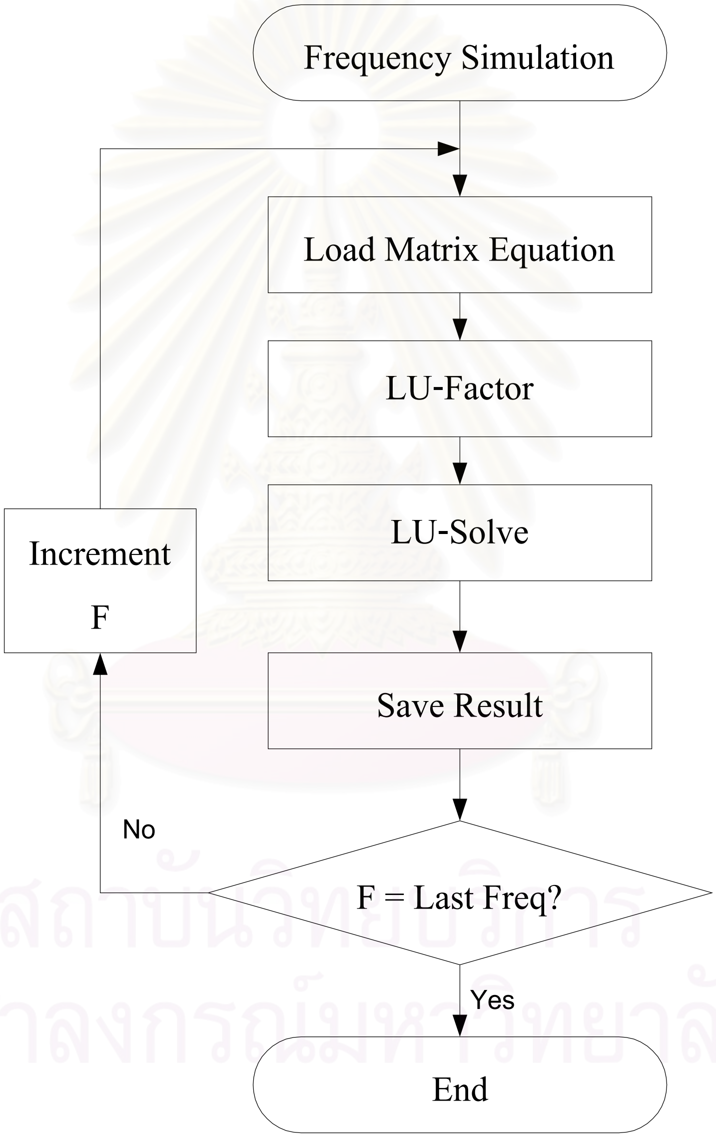
รูปที่ 3.1 แผนภูมิสายงานขั้นตอนวิธีการวิเคราะห์หาจุดทำงานสงบของโปรแกรม”เล็ก 6.0”

จากรูป 3.1 จะเห็นว่าโปรแกรม “เล็ก 6.0” แบ่งขั้นตอนการทำงานหลักๆออกเป็น 4 ขั้นตอนคือ

1. *Load Matrix Equation* คือการสร้างสมการเมทริกซ์ของวงจรไฟฟ้า ซึ่งโปรแกรม“เล็ก 6.0” ใช้วิธีโมดิฟายด์โนดัลในการสร้างสมการ $Ax = b$
2. *LU – Factor* คือขั้นตอนในการแยกตัวประกอบเมทริกซ์ A ให้เป็นเมทริกซ์ L และ U
3. *LU-Solve* คือขั้นตอนในการแก้สมการเมทริกซ์หาค่าผลเฉลยของวงจร
4. *Check Converge* คือขั้นตอนตรวจสอบว่าผลลัพธ์ที่ได้ลู่เข้าสู่คำตอบที่ถูกต้องหรือไม่

3.2 การวิเคราะห์ผลตอบสนองเชิงความถี่

การวิเคราะห์ผลตอบสนองเชิงความถี่ดังแสดงตามรูปที่ 3.2 มีขั้นตอนต่างจะการวิเคราะห์หาจุดทำงานสงบอยู่ที่ โปรแกรมจะต้องคำนวณหาคำตอบของสมการเมทริกซ์หลายครั้ง ณ จุดค่าความถี่ต่างๆ จึงจะสามารถวาดเป็นกราฟผลลัพธ์เชิงความถี่ได้ นอกจากนี้โปรแกรมจะต้องสร้างสมการเมทริกซ์ที่มีสมาชิกเป็นจำนวนเชิงซ้อน โดยที่คำตอบของวงจรจะถูกจัดอยู่ในเมทริกซ์ตัวแปร x เช่นเดิม



รูปที่ 3.2 แผนภูมิสายงานขั้นตอนวิธีการวิเคราะห์ผลตอบสนองเชิงความถี่ของโปรแกรม”เล็ก 6.0”

3.3 การวิเคราะห์ผลตอบสนองชั่วขณะ

สำหรับการวิเคราะห์ผลตอบสนองชั่วขณะเป็นการจำลองการทำงานของวงจรไฟฟ้าในทางเวลา มีขั้นตอนซับซ้อนกว่าการวิเคราะห์ผลทั้ง 2 แบบที่ผ่านมาเนื่องจากมีการประมาณสมการอนุพันธ์ให้เป็นสมการพีชคณิตโดยวิธี Backward Eular Formula ซึ่งมีหลักการคือแบ่งช่วงเวลาที่ต้องจำลองวงจรตั้งแต่เวลา 0 ถึงเวลา T นั้นออกเป็น N+1 จุดได้แก่ $t_0 < t_1 < t_2 < t_3 < \ldots < t_N$ โดยที่ $t_0 = 0$ และ $t_N = T$ และจุดเวลาเหล่านี้อยู่ห่างเท่าๆกัน เท่ากับ h ซึ่งเรียกว่า ขั้นเวลา (Time step) กล่าวคือ

$$t_1 - t_0 = t_2 - t_1 = t_3 - t_2 = \ldots = t_N - t_{N-1} = h = \frac{T}{N}$$

วิธีที่ใช้ในการแก้สมการอนุพันธ์ $\dot{x}(t) = f(x,t)$ ที่จุดเวลาต่างๆนี้ ใช้การประมาณอนุพันธ์ตามสูตรที่เรียกว่า Backward Euler Formula ดังสมการที่ (3.1)

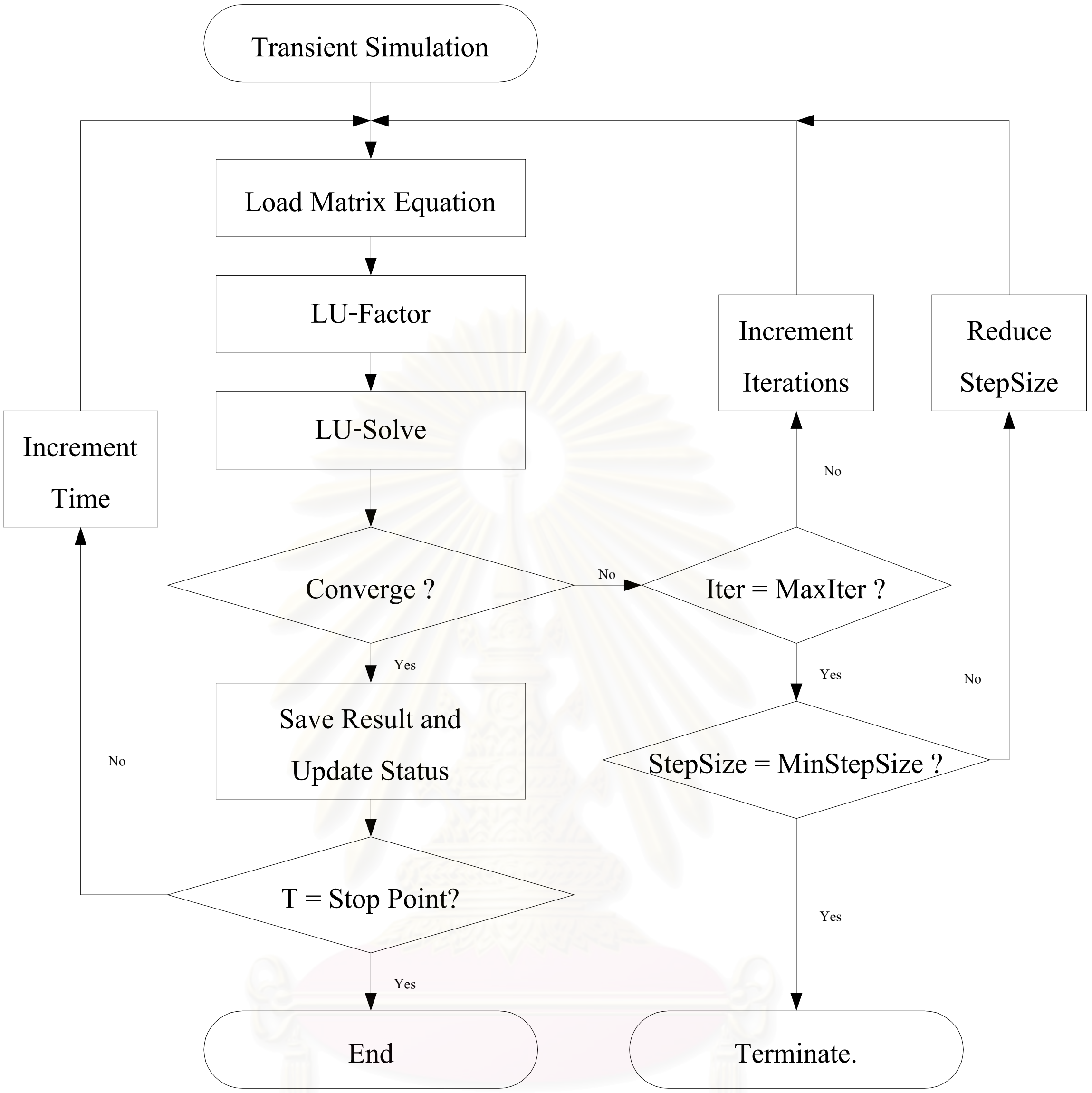
$$\dot{x}(t) \approx \frac{x(t_i) - x(t_{i-1})}{h} \qquad ; i = 1,2,3,\ldots,N \tag{3.1}$$

โดยการแทนอนุพันธ์ของตัวแปรทุกตัวในสมการวงจรด้วยการประมาณแบบ Backward Euler จะทำให้สมการอนุพันธ์ของวงจรเปลี่ยนเป็นสมการพีชคณิตล้วนๆ ตัวอย่างเช่น เมื่อมีตัวเก็บประจุในวงจรต่ออนุกรมกับตัวต้านทานจะทำให้เกิดสมการดังสมการที่ (3.2) หลังจากผ่านการประมาณอนุพันธ์ตามสูตรของ Backward Euler Formula แล้วจะกลายเป็นสมการพีชคณิตดังที่แสดงไว้ในสมการที่ (3.3)

$$C\frac{dV}{dt} = \frac{V}{R} \tag{3.2}$$

$$\frac{C}{h}\left[V(t_i) - V(t_{i-1})\right] = \frac{V}{R} \tag{3.3}$$

นอกจากนี้จะต้องมีการตรวจสอบการลู่เข้าของคำตอบที่คำนวณได้ด้วยวิธีการแยกตัวประกอบแอล-ยู ถ้าคำตอบที่ได้ลู่เข้า โปรแกรมจะปรับทันการ (update) ข้อมูล แล้วเลื่อนไปวิเคราะห์จุดเวลาถัดไปจนสิ้นสุดช่วงเวลาที่กำหนดให้ทำการจำลองวงจร ในทางตรงกันข้ามถ้าคำตอบที่ได้ไม่ลู่เข้าโปรแกรมจะทำการวนซ้ำ (Iteration) จนได้คำตอบที่ลู่เข้า แต่จะทำการวนซ้ำไม่เกินค่า MaxIter และถ้าทำการวนซ้ำจนครบ MaxIter ครั้งแล้วคำตอบยังไม่ลู่เข้า โปรแกรมจึงจะทำการลดขั้นเวลาที่ใช้ในการคำนวณลงครึ่งหนึ่งแล้วเริ่มทำการวนซ้ำใหม่อีกครั้ง แต่ถ้าการลดขั้นเวลาลงจนถึงค่า MinStepSize แล้วยังไม่ทำให้คำตอบลู่เข้า โปรแกรมจึงจะเลิกการคำนวณ ดังแสดงตามรูปที่ 3.3



รูปที่ 3.3 แผนภูมิสายงานขั้นตอนวิธีการวิเคราะห์ผลตอบสนองชั่วขณะของโปรแกรม"เล็ก 6.0"

บทที่ 4

การลดเวลาการคำนวณด้วยเทคนิคของเมทริกซ์มากเลขศูนย์

ในการคำนวณหาผลลัพธ์ตามทฤษฎีวงจรไฟฟ้าที่กล่าวไว้ในบทที่ 2 เราจะพบว่าส่วนใหญ่แล้วเมทริกซ์สัมประสิทธิ์ A ในสมการวงจร $A\,x = b$ มีคุณสมบัติเป็นเมทริกซ์มากเลขศูนย์ จึงสามารถนำเทคนิคเฉพาะสำหรับเมทริกซ์มากเลขศูนย์มาใช้ให้เกิดประโยชน์ในโปรแกรมจำลองวงจรไฟฟ้าได้ โดยจะแบ่งได้เป็น 4 เรื่องหลักคือ

1. การข้ามการคำนวณกับค่าศูนย์
2. โครงสร้างการเก็บข้อมูล (Data Structure)
3. การเรียงลำดับใหม่ (Reordering)
4. การกลับเศษส่วนค่าสมาชิกในแนวทแยงมุขสำคัญ

4.1 <u>การข้ามการคำนวณกับค่าศูนย์</u>

โดยปรกติ การแก้สมการเมทริกซ์ขนาด n x n มิติ ด้วยวิธีแยกตัวประกอบแอล-ยูตามที่กล่าวไว้แล้วในบทที่ 2 จะมีจำนวนครั้งการคูณในระดับ(Order) $O(n^3)$ จำนวนครั้งการหารระดับ $O(n^2)$ และจำนวนครั้งการบวกและลบระดับ $O(n^3)$ แต่เนื่องจากเมทริกซ์สัมประสิทธิ์ A นั้นมักเป็นเมทริกซ์มากเลขศูนย์เพราะสมาชิกส่วนใหญ่ในเมทริกซ์สัมประสิทธิ์ A มีค่าเป็นศูนย์ การแก้สมการเมทริกซ์โดยวิธีแยกตัวประกอบแอล-ยูจึงมีการคำนวณโดยการบวก,ลบ,คูณและหารกับสมาชิกที่มีค่าเป็นศูนย์เป็นจำนวนมาก ซึ่งเป็นการคำนวณที่ไม่จำเป็นเนื่องจาก

1. การบวกหรือลบด้วยศูนย์ จะได้ผลลัพธ์เป็นค่าเดิม
2. การคูณด้วยศูนย์ หรือ ศูนย์หารด้วยจำนวนใดๆ จะได้ผลลัพธ์เป็นศูนย์

ดังนั้นหากสามารถข้ามการคำนวณกับสมาชิกที่มีค่าเป็นศูนย์ไปได้ จะทำให้เหลือการคำนวณที่จำเป็นจริงๆไม่มาก โดยที่ยังได้ผลลัพธ์การแก้สมการเมทริกซ์ถูกต้องเหมือนเดิม

การข้ามการคำนวณกับค่าศูนย์ เป็นหัวใจสำคัญของวิทยานิพนธ์เรื่องนี้ เพราะเทคนิคต่างๆที่ได้เสนอในวิทยานิพนธ์ล้วนเป็นเทคนิคที่ช่วยให้การข้ามการคำนวณกับค่าศูนย์มีประสิทธิผลมากยิ่งขึ้น

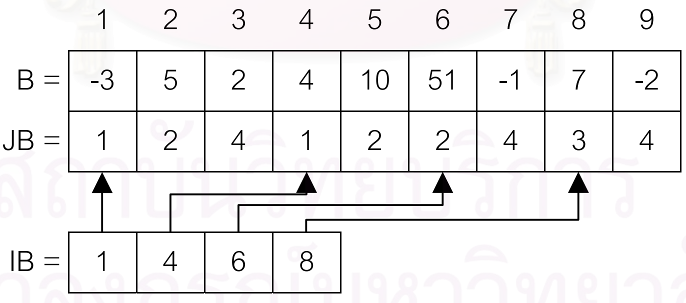
4.2 <u>โครงสร้างการเก็บข้อมูล</u>

มีโครงสร้างข้อมูลหลายแบบที่ออกแบบมาเพื่อเก็บข้อมูลเมทริกซ์มากเลขศูนย์โดยเฉพาะ โดยเน้นข้อดีที่จะใช้พื้นที่เก็บข้อมูลน้อยกว่าแบบปรกติซึ่งใช้พื้นที่เก็บข้อมูลจำนวน n^2 ช่อง สำหรับเมทริกซ์ขนาด n*n มิติ เพราะโครงสร้างข้อมูลส่วนใหญ่ที่ใช้เก็บข้อมูลเมทริกซ์มากเลขศูนย์ มักจะเลือกเก็บข้อมูลเฉพาะสมาชิกของเมทริกซ์ที่มีค่าไม่เป็นศูนย์เท่านั้น และจะไม่เก็บข้อมูลของสมาชิกที่มีค่าเป็นศูนย์ซึ่งมีจำนวนมากในเมทริกซ์ ทำให้ใช้พื้นที่เก็บข้อมูลน้อยกว่า

ตัวอย่างโครงสร้างข้อมูลแบบหนึ่งที่จะนำเสนอมีชื่อว่า Yale sparse matrix package [10] หรือ Compressed Sprase Row format (CSR) เป็นโครงสร้างข้อมูลที่เก็บข้อมูลเฉพาะค่าสมาชิกที่ไม่เป็นศูนย์ไว้ในแถวลำดับ(Array) B เรียงตามตำแหน่งในเมทริกซ์จากซ้ายไปขวาและบนลงล่าง เก็บตำแหน่งตามคอลัมน์ในแถวลำดับ JB และแถวลำดับ IB เก็บว่าสมาชิกที่ไม่เป็นศูนย์ตัวแรกในแต่ละแถวเก็บอยู่ที่ได้ในแถวลำดับ B เช่น ตัวอย่างเมทริกซ์มากเลขศูนย์ขนาด 4*4 มิติตามรูปที่ 4.1 สามารถเก็บข้อมูลลงในแถวลำดับ B, JB และ IB ได้ดังรูปที่ 4.2

$$\begin{bmatrix} -3 & 5 & 0 & 2 \\ 4 & 10 & 0 & 0 \\ 0 & 51 & 0 & -1 \\ 0 & 0 & 7 & -2 \end{bmatrix}$$

รูปที่ 4.1 ตัวอย่างเมทริกซ์มากเลขศูนย์ขนาด 4*4 มิติ



	1	2	3	4	5	6	7	8	9
B =	-3	5	2	4	10	51	-1	7	-2
JB =	1	2	4	1	2	2	4	3	4

IB =	1	4	6	8

รูปที่ 4.2 โครงสร้างข้อมูลตัวอย่างที่เก็บข้อมูลของเมทริกซ์ในรูปที่ 4.1

โครงสร้างข้อมูลตามรูปที่ 4.2 นั้นแม้จะใช้พื้นที่ในการเก็บข้อมูลเมทริกซ์น้อย แต่มีข้อเสียที่การค้นหาสมาชิกในคอลัมน์จะใช้เวลามาก เช่น การหาว่ามีสมาชิกตัวใดบ้างอยู่ในคอลัมน์ที่ 3 จะต้องเข้าไปค้นหาค่า 3 ในแถวลำดับ JB ทั้งหมด โครงสร้างข้อมูลนี้จึงไม่เหมาะที่จะใช้เก็บเมทริกซ์สัมประสิทธิ์ เพราะในการแยกตัวประกอบแอล-ยูจะต้องมีการค้นหาสมาชิกที่อยู่ในคอลัมน์เดียวกันทุกคอลัมน์ด้วย

โครงสร้างข้อมูลที่เหมาะสำหรับการแก้สมการเมทริกซ์ด้วยวิธีแยกตัวประกอบแอล-ยูจึงควรมีคุณสมบัติดังนี้

1. สามารถเก็บเมทริกซ์ที่ไม่สมมาตรได้ เนื่องจากก่อนการแยกตัวประกอบแอล-ยูจะทำการเรียงลำดับใหม่ซึ่งจะทำให้ได้เมทริกซ์ที่ไม่สมมาตร
2. เก็บข้อมูลทั้งหมดไว้ในแถวลำดับและเชื่อมความสัมพันธ์ด้วยดัชนี (index) โดยไม่ใช้ตัวชี้ (Pointer) เนื่องจากตัวชี้ใช้พื้นที่หน่วยความจำมากกว่าดัชนีถึง 2 เท่า (ดัชนี 1 ตัวใช้หน่วยความจำ 2 ไบต์, ตัวชี้ 1 ตัวใช้หน่วยความจำ 4 ไบต์)
3. ข้อมูลที่เก็บในแถวลำดับไม่จำเป็นต้องเรียงลำดับตามตำแหน่งที่อยู่ในเมทริกซ์ เนื่องจากการใส่ข้อมูลลงในเมทริกซ์สัมประสิทธิ์ตามขั้นตอน Load Matrix Equation ในโปรแกรม ”เล็ก 6.0” จะไม่เรียงลำดับตามตำแหน่งที่อยู่ในเมทริกซ์เช่นกัน
4. สามารถเพิ่มสมาชิกใหม่ที่มีค่าไม่เป็นศูนย์เข้ามาในเมทริกซ์ได้โดยไม่ต้องแทรกข้อมูลเพิ่มในแถวลำดับ แต่ให้เพิ่มข้อมูลสมาชิกใหม่ต่อท้ายแถวลำดับแทน ซึ่งใช้เวลาในการเพิ่ม Fill-Ins น้อยกว่า
5. ค้นหาสมาชิกที่อยู่ในแถวเดียวกันโดยเรียงลำดับจากคอลัมน์แรกไปคอลัมน์สุดท้าย หรือค้นหาสมาชิกที่อยู่ในคอลัมน์เดียวกันโดยเรียงลำดับจากแถวแรกไปแถวสุดท้ายได้อย่างรวดเร็ว จะทำให้ใช้เวลาในการแยกตัวประกอบแอล-ยูน้อยลง
6. สามารถสลับสมาชิกในแถวทั้งแถวหรือคอลัมน์ทั้งแนวได้ง่าย
7. เข้าถึงสมาชิกตัวที่อยู่ในแนวทแยงมุขสำคัญได้อย่างรวดเร็ว

จากคุณสมบัติทั้ง 7 ข้อที่ต้องการ จึงได้ออกแบบโครงสร้างข้อมูลที่ดัดแปลงมาจากโครงสร้างข้อมูลแบบรายการเชื่อมโยง (Link List) โดยสมาชิกที่มีค่าไม่เป็นศูนย์ตัวหนึ่งจะเก็บข้อมูลตามที่แสดงในรูปที่ 4.3 ดังนี้

Value	สำหรับเก็บค่าของสมาชิกที่ไม่เป็นศูนย์ในเมทริกซ์
Row	สำหรับเก็บเลขตำแหน่งแถวของข้อมูล Value
Col	สำหรับเก็บเลขตำแหน่งคอลัมน์ของข้อมูล Value
NextRow	เป็นดัชนีที่ชี้ไปยังข้อมูลแถวถัดไปที่อยู่ในคอลัมน์เดียวกับ Value
NextCol	เป็นดัชนีที่ชี้ไปยังข้อมูลคอลัมน์ถัดไปที่อยู่ในแถวเดียวกับ Value

Value	Row	Col	Next Row	Next Col

รูปที่ 4.3 ตัวอย่างโครงสร้างข้อมูล

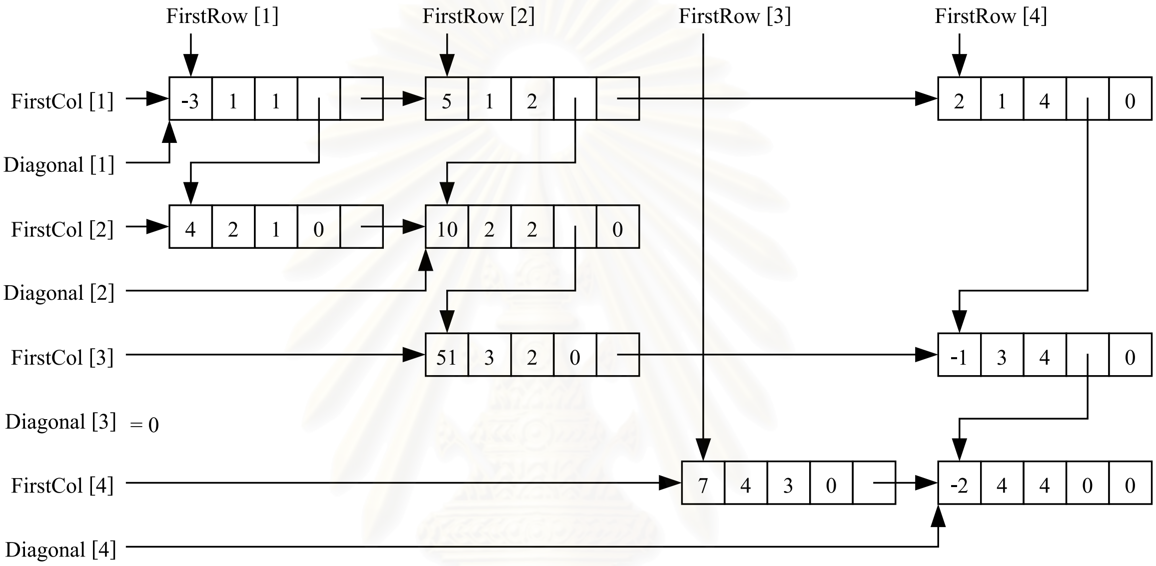
นอกจากนี้ ยังมีตัวแปร

FirstRow เป็นดัชนีที่ชี้ไปยังข้อมูลแถวแรกสุดในแต่ละคอลัมน์ของเมทริกซ์

FirstCol เป็นดัชนีที่ชี้ไปยังข้อมูลคอลัมน์แรกสุดในแต่ละแถวของเมทริกซ์

Diagonal เป็นดัชนีที่ชี้ไปยังข้อมูลแต่ละตัวที่เรียงอยู่ในแนวทะแยงมุมของเมทริกซ์

สำหรับเมทริกซ์มากเลขศูนย์ในรูปที่ 4.1 สามารถเก็บด้วยโครงสร้างข้อมูลนี้ดังแสดงในรูปที่ 4.4 ดัชนีที่มีค่าเป็น 0 จะแสดงว่าไม่ได้ชี้ไปที่ข้อมูลใดเลย ถือเป็นการสิ้นสุดรายการเชื่อมโยงนั้น



รูปที่ 4.4 โครงสร้างข้อมูลสำหรับเก็บข้อมูลของเมทริกซ์มากเลขศูนย์ในรูปที่ 4.1

	1	2	3	4	5	6	7	8	9
Value =	10	4	2	-2	7	-3	51	-1	5
Row =	2	2	1	4	4	1	3	3	1
Col =	2	1	4	4	3	1	2	4	2
NextRow =	7	0	8	0	0	2	0	4	1
NextCol =	0	1	0	0	4	9	8	0	3
FirstRow =	6	9	5	3					
FirstCol =	6	2	7	5					
Diagonal =	6	1	0	4					

รูปที่ 4.5 ข้อมูลที่เก็บในแถวลำดับต่างๆสำหรับการข้อมูลของเมทริกซ์ในรูปที่ 4.1

โครงสร้างข้อมูลแบบที่มีดัชนีชี้ไปยังค่าสมาชิกถัดไปที่มีค่าไม่เป็นศูนย์แบบนี้จะช่วยให้การคำนวณตามวิธีแยกตัวประกอบแอล-ยูทำได้ง่ายและรวดเร็วขึ้น ดังนี้

ขั้นตอนแยกตัวประกอบแอล-ยู ตามสูตรคำนวณ

$$l_{ik} = a_{ik} - \sum_{m=1}^{k-1} l_{im} u_{mk} \qquad \text{โดย } i \geq k \qquad (4.1)$$

$$u_{ki} = \left(a_{kj} - \sum_{m=1}^{k-1} l_{km} u_{mj} \right) \Big/ l_{kk} \qquad \text{โดย } j > k \qquad (4.2)$$

จะพบว่ามีการบวกของผลคูณระหว่างสมาชิกแถวที่ i และคอลัมน์ที่ j เป็นคู่ๆเรียงกันตามลำดับ ซึ่งการคูณกันนี้สามารถทำได้ตามขั้นตอนต่อไปนี้

```
mu = FirstRow[j]                          ; ดัชนี mu ชี้ไปยังสมาชิกแถวแรกในคอลัมน์ที่ j
ml = FirstCol[i]                          ; ดัชนี ml ชี้ไปยังสมาชิกคอลัมน์แรกในแถวที่ i
Sigma = 0                                 ; กำหนดค่าผลรวมเริ่มต้นให้มีค่าเป็น 0
While (Row[mu]<a) and (Col[ml]<a) do      ; หากสมาชิกที่ mu อยู่ในแถวที่น้อยกว่า a  และ
begin                                       สมาชิกที่ ml อยู่ในคอลัมน์ที่น้อยกว่า a ให้ทำซ้ำ
                                            ดังนี้
  If Row[mu] = Col[ml] then               ; ถ้าแถวของสมาชิกที่ mu มีค่าเท่ากับคอลัมน์ของ
  begin                                     สมาชิกที่ ml ให้
    Sigma = Sigma + Value[mu]*Value[ml]   ; ผลการคูณสมาชิกที่ mu กับสมาชิกที่ ml ถูกรวม
                                            ในตัวแปร Sigma
    mu = NextRow[mu]                      ; ดัชนี mu ชี้ไปยังสมาชิกแถวถัดไปในคอลัมน์ที่ j
    ml = NextCol[ml]                      ; ดัชนี ml ชี้ไปยังสมาชิกคอลัมน์ถัดไปในแถวที่ i
  end Else begin
    While Row[mu] > Col[ml] do            ; หากแถวของสมาชิกที่ mu ยังมีค่ามากกว่าคอลัมน์
      ml = NextCol[ml]                      ของสมาชิกที่ ml ให้ ml ชี้ไปยังสมาชิกคอลัมน์ถัดไป
    While Row[mu] < Col[ml] do            ; หากแถวของสมาชิกที่ mu ยังมีค่าน้อยว่าคอลัมน์
      mu = NextRow[mu]                      ของสมาชิกที่ ml ให้ mu ชี้ไปยังสมาชิกแถวถัดไป
  end
end                                       ; เมื่อจบขั้นตอน จะได้ผลรวมของการคูณใน Sigma
```

โครงสร้างข้อมูลที่มี NextRow และ NextCol นี้จะช่วยให้สามารถนำค่าสมาชิกที่ไม่เป็นศูนย์คู่ถัดไปมาคูณกันได้ทันทีโดยไม่ต้องเสียเวลาค้นหาสมาชิกใหม่อีกครั้ง นอกจากนี้แถวลำดับ Diagonal ยังช่วยให้หาค่าสมาชิกในแนวทแยงมุมสำคัญเพื่อนำมาหารในสมการที่ (4.2) ได้อย่างรวดเร็วอีกด้วย

หน่วยความจำที่ใช้เก็บเมทริกซ์แบบทั่วไปมีขนาดเท่ากับ $8 * n^2$ ไบต์ โดยที่ n คือจำนวนตัวแปรอิสระในวงจร และ 8 คือขนาดหน่วยความจำสำหรับเก็บเลขจำนวนจริง 1 ค่า

ส่วนขนาดหน่วยความจำที่ใช้เก็บโครงสร้างข้อมูลของเมทริกซ์มากเลขศูนย์สามารถคำนวณได้โดยคำนวณขนาดหน่วยความจำที่ใช้เก็บแถวลำดับทั้ง 8 ตัว ได้แก่

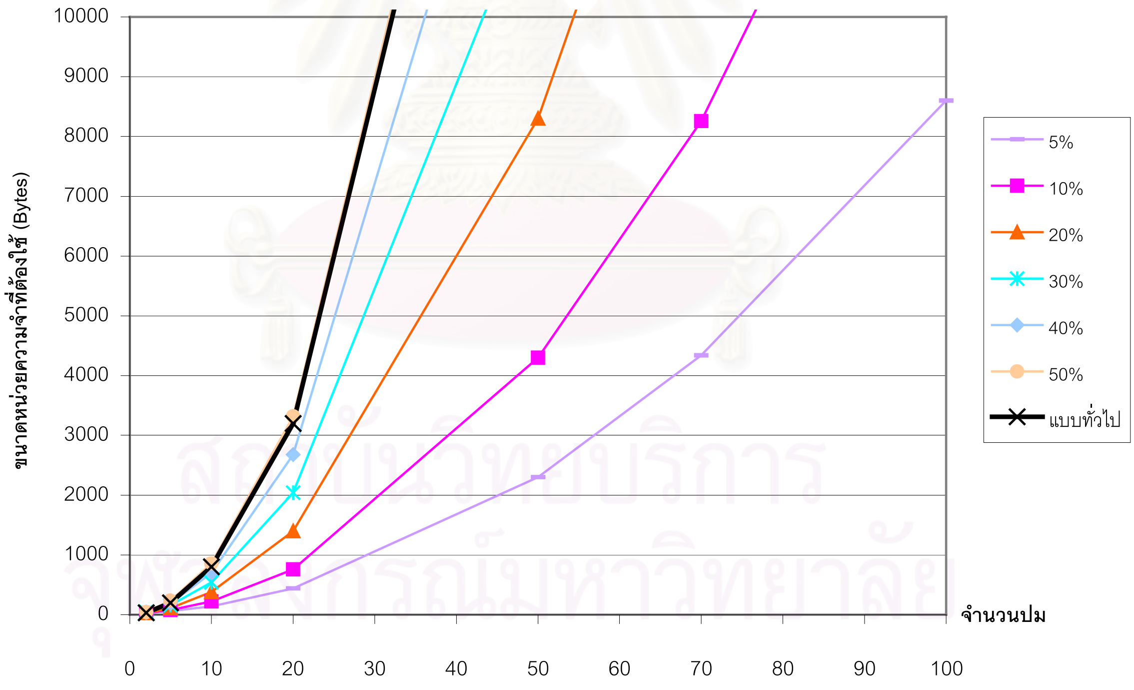
- Value มีขนาดเท่ากับ 8 * nzero ไบต์ โดยที่ nzero คือจำนวนสมาชิกที่ไม่เป็นศูนย์ในเมทริกซ์
- Row, Col, NextRow, NextCol แต่ละแถวลำดับมีขนาดเท่ากับ 2 * nzero ไบต์ โดยที่ 2 คือขนาดหน่วยความจำสำหรับเก็บค่าดัชนี 1 ตัว
- FirstRow, FirstCol, Diagonal แต่ละแถวลำดับมีขนาดเท่ากับ 2 * n ไบต์

รวมแล้วจะได้ว่าขนาดหน่วยความจำเท่ากับ (16*nzero)+(6*n) ไบต์

ถ้านิยามคำว่า “ดัชนีมากเลขศูนย์” (Sparsity Index) ดังนี้

$$\text{ดัชนีมากเลขศูนย์} = \frac{\text{จำนวนสมาชิกที่มีค่าไม่เป็นศูนย์ (nzero)}}{\text{จำนวนสมาชิกทั้งหมดในเมทริกซ์ } (n^2)} \times 100\%$$

จะได้ว่าโครงสร้างข้อมูลของเมทริกซ์มากเลขศูนย์มีขนาดเท่ากับ $(16*si*n^2) + (6n)$ ไบต์ โดยที่ si คือค่าดัชนีมากเลขศูนย์ โครงสรังข้อมูลแบบนี้จึงมีขนาดขึ้นกับจำนวนตัวแปรอิสระในวงจรและดัชนีมากเลขศูนย์เป็นสำคัญ ดังแสดงในรูปที่ 4.6



รูปที่ 4.6 กราฟความสัมพันธ์ระหว่างขนาดหน่วยความจำที่ใช้เก็บโครงสร้างข้อมูลของเมทริกซ์มากเลขศูนย์เทียบกับจำนวนตัวแปรอิสระในวงจร ที่ดัชนีมากเลขศูนย์ค่าต่างๆกัน

จะเห็นว่าถ้าดัชนีมากเลขศูนย์มีค่าน้อย โครงสร้างข้อมูลนี้จะใช้หน่วยความจำน้อยกว่าการเก็บแบบปรกติ โดยเฉพาะเมื่อมีจำนวนตัวแปรอิสระมากๆ ซึ่งโดยส่วนใหญ่วงจรไฟฟ้าที่ยิ่งมีจำนวนตัวแปรอิสระมากมักจะมีดัชนีมากเลขศูนย์น้อยลงอยู่แล้ว ดังนั้นการใช้โครงสร้างข้อมูลนี้เก็บเมทริกซ์สัมประสิทธิ์ของสมการวงจรประหยัดหน่วยความจำมากกว่า

การที่โครงสร้างข้อมูลใช้หน่วยความจำน้อยลง จะทำให้มีหน่วยความจำเหลือสำหรับให้โปรแกรมไว้ใช้ทำการจำลองวงจรไฟฟ้าที่มีจำนวนปมมากขึ้นหรือมีความซับซ้อนมากขึ้นได้

4.3 การเรียงลำดับใหม่ (Reordering)

การเรียงลำดับใหม่ คือ การสลับแถวและคอลัมน์ของเมทริกซ์สัมประสิทธิ์เพื่อย้ายตำแหน่งสมาชิกที่ต้องการให้ไปอยู่ในแนวทแยงมุขสำคัญ โดยสมาชิกของเมทริกซ์ที่อยู่ในแนวทแยงมุขสำคัญเรียกว่า Pivot

4.3.1 ประโยชน์ของการเรียงลำดับใหม่

จุดประสงค์ในการเรียงลำดับใหม่สำหรับงานวิทยานิพนธ์นี้ คือ ต้องการลดจำนวนครั้งของการคำนวณในการแยกตัวประกอบแอล-ยูลง ดังตัวอย่างเมทริกซ์ในรูปที่ 4.7

หากเมทริกซ์สัมประสิทธิ์มีสมาชิก $a_{23} = a_{32} = 0$ จะต้องทำการคูณ/หารจำนวน 8 ครั้ง และทำการลบจำนวน 5 ครั้งในขั้นตอนการแยกตัวประกอบแอล-ยู

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & 0 \\ a_{31} & 0 & a_{33} \end{bmatrix} \qquad L:U = \begin{bmatrix} l_{11} = a_{11} & u_{12} = a_{12}/l_{11} & u_{13} = a_{13}/l_{11} \\ l_{21} = a_{21} & l_{22} = a_{22} - l_{21}u_{12} & u_{23} = (0 - l_{21}u_{13})/l_{22} \\ l_{31} = a_{31} & l_{32} = 0 - l_{31}u_{12} & l_{33} = a_{33} - l_{31}u_{13} - l_{32}u_{23} \end{bmatrix}$$

รูปที่ 4.7 เมทริกซ์ตัวอย่าง และการคำนวณเพื่อแยกตัวประกอบแอล-ยู

แต่หากเพียงสลับแถวที่ 1 กับแถวที่ 3 และสลับคอลัมน์ที่ 1 กับคอลัมน์ที่ 3 จะทำให้เหลือการคูณ/หารเพียง 4 ครั้ง และเหลือการลบ 2 ครั้งเท่านั้น

$$\begin{bmatrix} a_{33} & 0 & a_{31} \\ 0 & a_{22} & a_{21} \\ a_{13} & a_{12} & a_{11} \end{bmatrix} \qquad L:U = \begin{bmatrix} l_{11} = a_{33} & u_{12} = 0 & u_{13} = a_{31}/l_{11} \\ l_{21} = 0 & l_{22} = a_{22} & u_{23} = a_{21}/l_{22} \\ l_{31} = a_{13} & l_{32} = a_{12} & l_{33} = a_{11} - l_{31}u_{13} - l_{32}u_{23} \end{bmatrix}$$

รูปที่ 4.8 เมทริกซ์ตัวอย่างหลังการ Reordering และการคำนวณเพื่อแยกตัวประกอบแอล-ยู

นอกจากการเรียงลำดับใหม่จะมีประโยชน์ในด้านการลดจำนวนครั้งในการคำนวณแล้ว ยังเป็นการจัดให้สมาชิกในแนวทแยงมุขสำคัญมีค่าไม่เป็นศูนย์ ทำให้ลดความผิดพลาดของโปรแกรมเนื่องจากการหารค่าด้วยศูนย์ (Division by zero) อีกด้วย

4.3.2 ประเภทของการเรียงลำดับใหม่

การเรียงลำดับใหม่มีอยู่หลายวิธี ซึ่งเราจะวัดประสิทธิภาพของแต่ละวิธีได้โดยดูจากจำนวนสมาชิกใหม่ที่มีค่าไม่เป็นศูนย์ที่เกิดขึ้นหลังจากการแยกตัวประกอบแอล-ยู หรือที่เรียกว่า Fill-ins โดยวิธีการเรียงลำดับใหม่ที่ดีจะทำให้เกิด Fill-Ins น้อยที่สุด ซึ่งมีแนวโน้มที่จะทำให้เกิดจำนวนครั้งการคำนวณน้อยที่สุดด้วย

การเรียงลำดับใหม่สามารถแบ่งได้เป็น 3 ประเภทดังนี้คือ

1. Greedy Algorithm เป็นวิธีง่ายที่สุด โดยเรียงลำดับแถวตามจำนวนสมาชิกที่ไม่เป็นศูนย์ในแถวนั้น โดยให้แถวแรกสุดมีจำนวนสมาชิกที่ไม่เป็นศูนย์น้อยที่สุด

2. Minimum Degree Rule เป็นวิธีเลือก Pivot ทีละตัวตามสูตรคำนวณที่กำหนด เมื่อเลือกได้แล้วจะคำนวณตำแหน่ง Fill-ins ที่จะเกิดขึ้น แล้วจึงเลือก Pivot ตัวถัดไปทีละตัวจนครบทั้งหมด ตัวอย่างของการเรียงลำดับใหม่แบบ Minimum Degree Rule นี้คือ Markowitz Method[11]

3. Minimum Local Fill-in เป็นวิธีเลือก Pivot ตัวที่จะทำให้เกิด Fill-ins น้อยที่สุดในสถานะนั้น เมื่อเลือกได้แล้วจะคำนวณตำแหน่ง Fill-ins ที่จะเกิดขึ้น แล้วจึงเลือก Pivot ตัวถัดไปทีละตัวเช่นเดียวกับวิธี Minimum Degree Rule ตัวอย่างของการเรียงลำดับใหม่แบบ Minimum Local Fill-in นี้คือ Berry Method[12]

วิธี Greedy Algorithm เป็นที่ใช้เวลาเลือก Pivot น้อยที่สุด แต่มีข้อเสียคือจะทำให้เกิด Fill-Ins มากที่สุด ในขณะที่วิธี Minimum Local Fill-in จะทำให้เกิด Fill-Ins น้อยที่สุด แต่ต้องใช้เวลาเลือก Pivot มากที่สุด ส่วนวิธี Minimum Degree Rule เป็นวิธีที่ใช้เวลาเลือก Pivot ค่อนข้างน้อย แต่ยังทำให้เกิด Fill-Ins มากกว่าวิธี Minimum Local Fill-in เพียงเล็กน้อย ดังนั้นในงานวิทยานิพนธ์นี้จึงเลือกใช้วิธีเรียงลำดับใหม่แบบ Markowitz Method

4.3.3 Markowitz Method

Markowitz Method ถูกเสนอโดย H.M. Markowitz ในปี ค.ศ. 1957 เป็นวิธีการเรียงลำดับใหม่ประเภท Minimum Degree Rule มีข้อดีคือใช้เวลาในการเลือก Pivot น้อยแต่ยังสามารถเรียงลำดับเมทริกซ์ที่ทำให้เกิด Fill-Ins น้อยได้ โดยการทดสอบพบว่าการเรียงลำดับใหม่ตามวิธี Markowitz Method นั้นจะทำให้เกิด Fill-Ins มากกว่าวิธี Berry Method ประมาณ 5 % เท่านั้น [13] แต่ใช้เวลาในการเลือก Pivot น้อยกว่ามาก

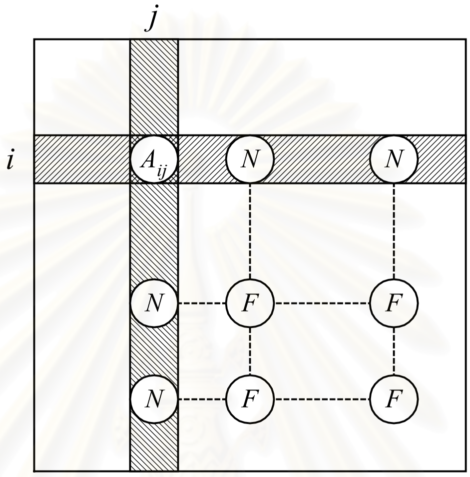
Markowitz ได้เสนอทฤษฎีไว้ว่า fill-in ที่จะเกิดขึ้นได้มากที่สุดหากเลือกสมาชิกแถวที่ i คอลัมน์ที่ j (A_{ij}) เป็น Pivot จะมีจำนวนเท่ากับ

$$Markowitz\ Measure\ (M_{ij}) \quad = \quad R_i \times C_j \tag{4.3}$$

โดยที่ M_{ij} คือ ค่า Markowitz Measure ของสมาชิกที่ไม่เป็นศูนย์ A_{ij}

R_i คือ จำนวนสมาชิกที่มีค่าไม่เป็นศูนย์ที่อยู่ในแถวที่ i โดยไม่นับสมาชิก A_{ij}

C_j คือ จำนวนสมาชิกที่มีค่าไม่เป็นศูนย์ที่อยู่ในคอลัมน์ที่ j โดยไม่นับสมาชิก A_{ij}



รูปที่ 4.9 ตัวอย่างการเกิด Fill-ins เมื่อเลือกสมาชิก A_{ij} เป็น Pivot

ดังตัวอย่างในรูปที่ 4.9 หากเลือกสมาชิก เป็น Pivot จะทำให้มีโอกาสเกิด Fill-In ได้อย่างมาก 4 ตัวเท่านั้น วิธี Markowitz Method จึงเลือก Pivot จากสมาชิกที่มีค่าไม่เป็นศูนย์ที่มีค่า Markowitz Measure น้อยที่สุดก่อนเพราะสมาชิกตัวนั้นจะมีโอกาสเกิด Fill-in ใหม่ได้จำนวนน้อยที่สุด จากนั้นจึงคำนวณหา Fill-ins ที่จะเกิดใหม่ แล้วคำนวณหาค่า Markowitz Measure ของแถวและคอลัมน์ที่เหลืออีกครั้งเพื่อหา Pivot ตัวถัดไปจนครบ

อย่างไรก็ตาม การเรียงลำดับเมทริกซ์ใหม่มีผลกระทบต่อความแม่นยำของผลลัพธ์เป็นอย่างมาก เพราะหากค่า Pivot มีค่าน้อยเกินไปจะทำให้ผลลัพธ์จากการคำนวณผิดพลาดไปจากคำตอบจริงมาก ซึ่งความคาดเคลื่อนนี้เรียกว่า Round-off Errors โดยเป็นผลมาจากการที่คอมพิวเตอร์ไม่สามารถเก็บเลขทศนิยมทั้งหมดของตัวเลขจำนวนจริงที่นำมาคำนวณได้ การเลือก Pivot จึงควรเลือก Pivot ที่มีค่าสัมบูรณ์มากที่สุด เพื่อลดผลความคาดเคลื่อนนี้ลง

ดังนั้น เราจึงกำหนดเกณฑ์ในการเลือก Pivot ดังนี้

1. เลือกสมาชิกที่มีค่าไม่เป็นศูนย์ที่มีค่า Markowitz Measure น้อยที่สุด
2. หากมีสมาชิกที่มีค่า Markowitz Measure น้อยที่สุดมากกว่า 1 ตัว ให้เลือกสมาชิกตัวที่มีค่าสัมบูรณ์มากที่สุด

3. หากมีสมาชิกที่มีค่า Markowitz Measure น้อยที่สุดและมีค่าสัมบูรณ์มากที่สุดมากกว่า 1 ตัวแล้ว ให้เลือกสมาชิกตัวที่มีค่า R_i น้อยที่สุด เพราะหากเลือกสมาชิกตัวนั้นเป็น Pivot จะต้องนำ Pivot ไปหารสมาชิกที่มีค่าไม่เป็นศูนย์ที่อยู่ในแถวเดียวกับ Pivot จำนวน R_i ตัว ดังนั้นจึงเลือกสมาชิกตัวที่มีค่า R_i น้อยที่สุด

4.4 การกลับเศษส่วนค่าสมาชิกในแนวทแยงมุขสำคัญ

การกลับเศษส่วนค่าสมาชิกในแนวทแยงมุขสำคัญเป็นเทคนิคอย่างหนึ่งที่ช่วยให้การคำนวณหาผลลัพธ์ได้รวดเร็วยิ่งขึ้น โดยในขั้นตอนการแยกตัวประกอบเมทริกซ์สัมประสิทธิ์ A ให้เป็นเมทริกซ์ L และ U เมื่อคำนวณค่าสมาชิกในแนวทแยงมุขสำคัญของเมทริกซ์ L (l_{ii}) ตามสมการที่ (4.1) ได้แล้ว ให้เก็บค่า $\frac{1}{l_{ii}}$ ลงในเมทริกซ์ L แทน และในการคำนวณคราวต่อไปที่ต้องการหารด้วยค่า l_{ii} จะให้นำค่า $\frac{1}{l_{ii}}$ ในเมทริกซ์ไปคูณแทน

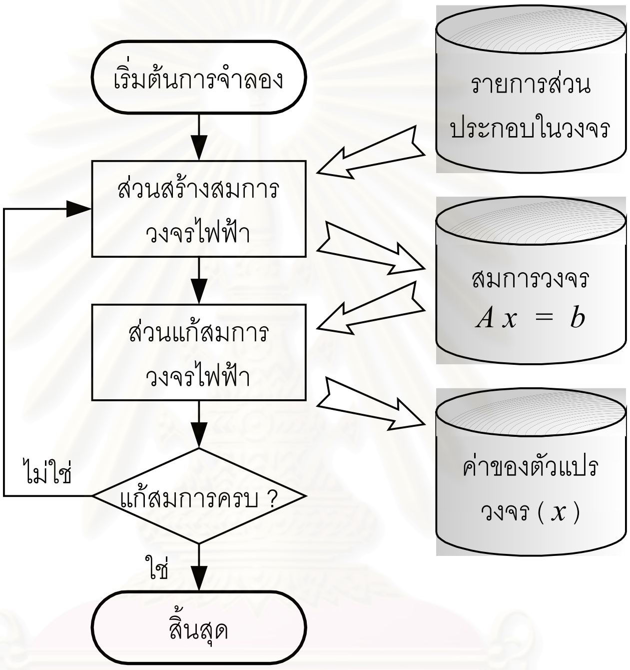
หากสังเกตตามขั้นตอนการแยกตัวประกอบแอล-ยูแล้วจะพบว่า หลังจากการคำนวณค่า l_{ii} ได้แล้ว ค่า l_{ii} นั้นจะถูกใช้เพียงการนำไปหารค่าสมาชิกตัวอื่นๆในเมทริกซ์เท่านั้น จึงไม่มีความจำเป็นที่จะต้องเก็บค่า l_{ii} ไว้ในเมทริกซ์แต่เก็บค่า$\frac{1}{l_{ii}}$ แทนได้

การที่เทคนิคนี้สามารถลดเวลาในการคำนวณลงได้เป็นเพราะหน่วยประมวลผลกลาง (CPU) ต้องใช้เวลาในการหารมากกว่าการคูณมาก โดยสำหรับหน่วยประมวลผลกลางระดับ Pentium ขึ้นมาจะใช้เวลาทำงานคำสั่ง FDIV ซึ่งเป็นคำสั่งหารเลขจำนวนจริงจำนวน 33 สัญญาณนาฬิกา(Clock) ส่วนคำสั่ง FMUL ซึ่งเป็นคำสั่งคูณเลขจำนวนจริงใช้เวลาทำงานเพียง 3 สัญญาณนาฬิกาเท่านั้น [13] อีกทั้งคำสั่ง FMUL ยังสามารถคำนวณแบบ Overlapping ได้ด้วย ดังนั้นเทคนิคนี้จึงเป็นการแปรเปลี่ยนการหารหลายๆครั้งให้เป็นการหารเพียงครั้งเดียวกับการคูณหลายๆครั้งแทน

บทที่ 5

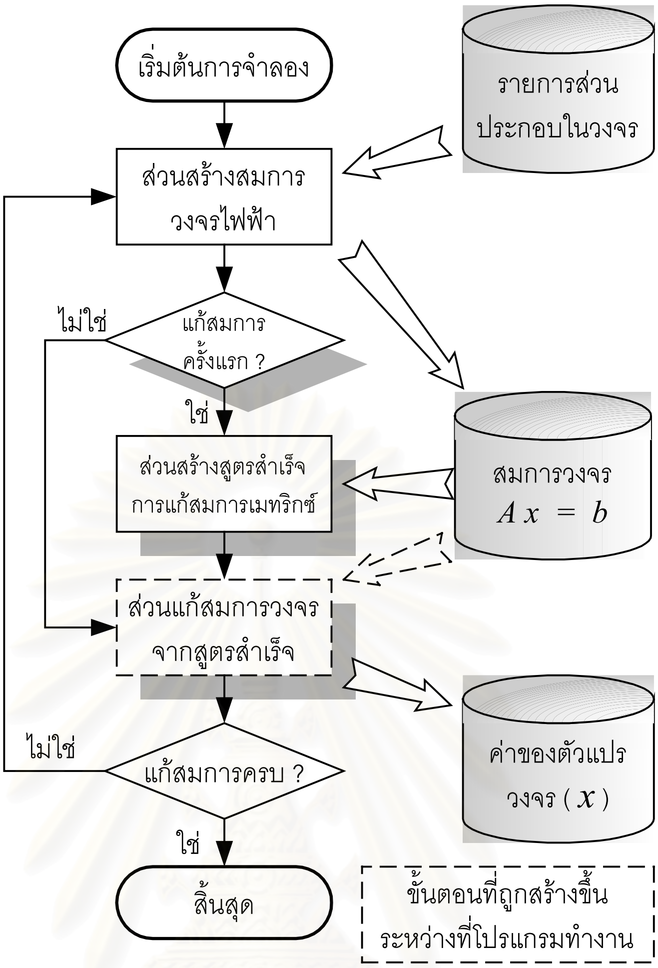
การสร้างสูตรสำเร็จการแก้สมการเมทริกซ์

การจำลองการทำงานของวงจรไฟฟ้าด้วยโปรแกรม “เล็ก 6.0” ในโมดการวิเคราะห์ผลตอบสนองเชิงความถี่และโมดการวิเคราะห์ผลตอบสนองชั่วขณะ มีขั้นตอนการทำงานของโปรแกรมตามรูปที่ 5.1



รูปที่ 5.1 ขั้นตอนการทำงานของโปรแกรมวิเคราะห์วงจรทั่วไป

โปรแกรม “เล็ก” จำเป็นต้องแก้สมการเมทริกซ์ซ้ำกันหลายสิบหลายร้อยครั้งเพื่อให้ได้ผลคำตอบที่จุดความถี่หรือจุดเวลาต่างๆ โดยที่สมการเมทริกซ์จะยังคงเป็นสมการลักษณะเดิม คือ มีขนาดมิติของเมทริกซ์และเวกเตอร์ตัวแปร x คงเดิม ต่างกันเพียงค่าสมาชิกที่ไม่เป็นศูนย์ในเวกเตอร์คงที่ b และเมทริกซ์สัมประสิทธิ์ A ที่จะเปลี่ยนแปลงตามสถานะของวงจร ดังนั้นถ้าเปรียบให้สมาชิกที่มีค่าไม่เป็นศูนย์ในเมทริกซ์เสมือนตัวแปรในสมการ จะสามารถสร้างสูตรสำเร็จการแก้สมการเมทริกซ์ขึ้นมาได้ก่อนในการแก้สมการครั้งแรกและเมื่อต้องการคำนวณผลคำตอบที่จุดความถี่หรือจุดเวลาถัดไปๆ จะสามารถใช้สูตรสำเร็จการแก้สมการเมทริกซ์นี้แก้สมการหาผลคำตอบได้อย่างรวดเร็วกว่าการที่ต้องใช้วิธีแยกตัวประกอบแอล-ยูแก้สมการใหม่อีกครั้ง ดังแสดงขั้นตอนการทำงานในรูปที่ 5.2



รูปที่ 5.2 ขั้นตอนการทำงานของโปรแกรมเมื่อใช้เทคนิคการสร้างสูตรสำเร็จการแก้สมการเมทริกซ์

ในงานวิทยานิพนธ์นี้ได้สร้างสูตรสำเร็จการแก้สมการเมทริกซ์ขึ้นมา 2 แบบ ได้แก่

1. การสร้างรหัสแบบแปลคำสั่ง (Interpretive Code Generation)
2. การสร้างรหัสคำสั่งเครื่อง (Machine Code Generation)

5.1 <u>การสร้างรหัสแบบแปลคำสั่ง</u>

รหัสแบบแปลคำสั่ง เป็นชุดคำสั่งง่ายๆที่บอกถึงวิธีแก้สมการเมทริกซ์ของวงจรไฟฟ้า โดยรหัสแบบแปลคำสั่งนี้จะแตกต่างกันแล้วแต่ลักษณะของวงจรไฟฟ้าที่โปรแกรมได้ทำการคำนวณหาค่าผลลัพธ์ของวงจรไฟฟ้านั้นๆ การสร้างรหัสแบบแปลคำสั่งจะทำขึ้นในขณะที่โปรแกรมกำลังแก้สมการเมทริกซ์ในครั้งแรก และเมื่อต้องแก้สมการเมทริกซ์นี้ในคราวต่อมา จะให้มาอ่านรหัสแบบแปลคำสั่งนี้แล้วปฏิบัติตามคำสั่ง ซึ่งจะได้ผลลัพธ์การคำนวณถูกต้องเช่นเดียวกับการแก้สมการเมทริกซ์ใหม่

5.1.1 รูปแบบของรหัสแบบแปลคำสั่ง

ในการทำวิทยานิพนธ์ครั้งนี้ ได้กำหนดรหัสแบบแปลคำสั่งขึ้นมา 4 แบบที่สามารถเรียงต่อกันเป็นสูตรสำเร็จการแก้สมการเมทริกซ์ได้อย่างสมบูรณ์ รหัสแบบแปลคำสั่งทั้งหมดจะถูกเก็บเรียงต่อกันไปเรื่อยๆในแถวลำดับ Inp_Code โดยรหัสแบบแปลคำสั่ง 4 แบบคือ

5.1.1.1 กำหนดค่า Fill-in เป็นศูนย์

คือการกำหนดค่าสมาชิกในเมทริกซ์ที่ตำแหน่งดัชนี a ให้มีค่าเป็นศูนย์ ตามสมการที่ (5.1) คำสั่งนี้จะใช้เป็นคำสั่งแรกสุดเพื่อกำหนดค่าสมาชิกที่เป็น Fill-in ใหม่ให้มีค่าเป็นศูนย์ก่อนเริ่มคำนวณ

$$A[a] = 0 \tag{5.1}$$

การเก็บรหัสคำสั่งเครื่องจะเก็บลงในแถวลำดับ Inp_Code โดยข้อมูลตัวแรกจะเป็น -2 และตามตัวค่าดัชนี b ตามรูปแบบในรูปที่ 5.3

-2	Index a	0

รูปที่ 5.3 รูปแบบการเก็บรหัสแบบแปลคำสั่ง”กำหนดค่า Fill-In เป็นศูนย์”ลงในแถวลำดับ Inp_Code

5.1.1.2 *การคูณแล้วลบ (Multiplication - Subtraction)*

คือการคูณค่าสมาชิกในเมทริกซ์ที่ตำแหน่งดัชนี b และ c แล้วนำมาลบออกจากค่าสมาชิกที่ตำแหน่งดัชนี a และเก็บค่าผลลัพธ์ที่ตำแหน่งดัชนี a ดังแสดงในสมการที่ (5.2)

$$A[a] = A[a] - (A[b] * A[c]) \tag{5.2}$$

ส่วนการเก็บรหัสแบบแปลคำสั่งลงในแถวลำดับ Inp_Code จะเก็บตัวเลขคำสั่งละ 3 ตัวตามรูปแบบในรูปที่ 5.4

Index a	Index b	Index c

รูปที่ 5.4 รูปแบบการเก็บรหัสแบบแปลคำสั่ง”การคูณแล้วลบ”ลงในแถวลำดับ Inp_Code

5.1.1.3 การกลับเศษส่วน

คือการกลับค่าเศษส่วนของสมาชิกในเมทริกซ์ที่ตำแหน่งดัชนี a ซึ่งจะใช้กับสมาชิกในแนวทแยงมุขสำคัญในเมทริกซ์ A ตามเทคนิคการกลับเศษส่วนค่าสมาชิกในแนวทแยงมุขสำคัญ (หัวข้อ 4.4) คำสั่งนี้แสดงได้ดังสมการที่ (5.3)

$$A[a] = 1 / A[a] \tag{5.3}$$

การเก็บรหัสคำสั่งเครื่องจะเก็บลงในแถวลำดับ Inp_Code โดยข้อมูลตัวแรกจะเป็น -1 และตามด้วยค่าดัชนี b ตามรูปแบบในรูปที่ 5.5

-1	Index a	0

รูปที่ 5.5 รูปแบบการเก็บรหัสแบบแปลคำสั่ง”การกลับเศษส่วน”ลงในแถวลำดับ Inp_Code

5.1.1.4 การคูณ (Multiplication)

คือการคูณระหว่างค่าสมาชิกในเมทริกซ์ที่ตำแหน่งดัชนี a กับค่าสมาชิกที่ตำแหน่งดัชนี d แล้วเก็บค่าผลลัพธ์ที่ตำแหน่งดัชนี a ดังแสดงในสมการที่ (5.4) คำสั่งนี้จะใช้แทนการหารทั่วไปตามสูตรการหาค่า u_{kj} และ z_i (แสดงไว้ในบทที่ 2) โดยสมาชิกที่ตำแหน่งดัชนี d จะเป็นสมาชิกในแนวทแยงมุมสำคัญซึ่งถูกกลับเศษเป็นส่วนด้วยรหัสแบบแปลคำสั่ง”การกลับเศษส่วน”แล้ว

$$A[a] = A[a] * A[d] \qquad (5.4)$$

การเก็บรหัสแบบแปลคำสั่งลงในแถวลำดับ Inp_Code จะเก็บตัวเลขแรกด้วยค่า 0 และตามมาด้วยค่าดัชนี a และ d ตามลำดับ ดังรูปที่ 5.6

0	Index a	Index d

รูปที่ 5.6 รูปแบบการเก็บรหัสแบบแปลคำสั่ง”การหาร”ลงในแถวลำดับ Inp_Code

แถวลำดับ Inp_Code สามารถเปลี่ยนแปลงขนาดความยาวได้ตามลักษณะของวงจรที่นำมาวิเคราะห์ โดยเริ่มแรกจะกำหนดความยาวของแถวลำดับไว้ n^2 คำสั่ง (n คือจำนวนตัวแปรอิสระในสมการวงจร) แต่ถ้าหากรหัสแบบแปลคำสั่งมีขนาดใหญ่กว่านี้ โปรแกรมจะจองแถวลำดับ Inp_Code ใหม่ที่มีขนาดใหญ่ขึ้นแล้วย้ายรหัสแบบแปลคำสั่งมาเก็บไว้ที่แถวลำดับใหม่นี้ และคืนพื้นที่หน่วยความจำของ Inp_Code เดิมกลับให้กับระบบ

การตีความหมายของรหัสแบบแปลคำสั่งที่เก็บเรียงกันอยู่ในแถวลำดับ Inp_Code จะทำโดยการอ่านค่าในแถวลำดับออกมาทีละ 3 ตัว แล้วดูค่าในตัวแรกว่าเป็น -2, -1, 0 หรือค่าจำนวนเต็มบวก เนื่องจากค่าดัชนีทั้งหมดจะเป็นเลขจำนวนเต็มบวก ดังนั้นไม่มีปัญหาเรื่องการตีความหมายผิดคำสั่งจากคำสั่ง”การคูณแล้วลบ” ไปเป็นคำสั่งอื่น

5.1.2 วิธีใช้งานรหัสแบบแปลคำสั่ง

เมื่อได้รหัสแบบแปลคำสั่งที่สมบูรณ์ในแถวลำดับ Inp_Code แล้ว และต้องการใช้รหัสแบบแปลคำสั่งนี้เพื่อแก้สมการเมทริกซ์ โปรแกรมจะต้องอ่านข้อมูลออกมาจากแถวลำดับ Inp_Code ทีละ 3 ตัวเลข และตีความหมายของรหัสจากตัวเลขตัว จากนั้นจึงคำนวณตามรหัสแบบแปลคำสั่งนั้น

เมื่อคำนวณตามรหัสแบบแปลคำสั่งครบทั้งหมด จะได้ผลคำตอบในแถวลำดับที่เก็บค่าสมาชิกของเมทริกซ์ b แต่ค่าคำตอบจะยังไม่เรียงลำดับตรงกับตัวแปรวงจรที่กำหนด เนื่องจากเทคนิคการเรียงลำดับใหม่ทำให้มีการสลับลำดับของตัวแปรวงจรก่อนการคำนวณ ดังนั้นเมื่อคำนวณตามรหัสแบบ

แปลคำสั่งครบทั้งหมดแล้วจะต้องสลับค่าในเมทริกซ์ b ให้ถูกต้องก่อน จึงจะนำผลคำตอบไปแสดงเป็นผลลัพธ์ต่อไป

5.2 <u>การสร้างรหัสคำสั่งเครื่อง (Machine Code Generating)</u>

การใช้รหัสแบบแปลคำสั่งแม้จะเป็นวิธีที่ง่ายและมีประสิทธิภาพดี แต่ยังจำเป็นต้องใช้เวลาในการตีความหมายของรหัสแล้วจึงคำนวณตามรหัสนั้น ดังนั้นเปลี่ยนจากการสร้างรหัสแบบแปลคำสั่งมาเป็นการสร้างรหัสคำสั่งเครื่องแทนเพื่อต้องการลดเวลาการตีความหมายของรหัสทิ้งไป วิธีนี้เครื่องคอมพิวเตอร์จะสามารถคำนวณผลลัพธ์ได้ทันทีโดยไม่ต้องตีความหมายของรหัสอีก ทำให้สามารถแก้สมการเมทริกซ์ได้โดยใช้เวลาน้อยที่สุด

5.2.1 รหัสคำสั่งเครื่องที่ใช้คำนวณเลขจำนวนจริง

ในการทำวิจัยครั้งนี้ได้สร้างรหัสคำสั่งเครื่องโดยอิงกับหน่วยประมวลผลกลาง (CPU:Central Processing Unit) ตระกูล Pentium ของบริษัท Intel เท่านั้น รหัสคำสั่งเครื่องที่เกี่ยวข้องกับการคำนวณเลขจำนวนจริงโดย FPU (Floating Point Unit) จะขึ้นต้นตัวอักษรด้วย ‘F’ แบ่ง ได้เป็นกลุ่มได้ 6 กลุ่มดังนี้ [14]

1. *Data Transfer* เป็นคำสั่งอ่าน, เขียน, สลับและย้ายค่าจาก Register
2. *Basic Arithmetic* เป็นคำสั่งคำนวณพื้นฐาน
3. *Comparison* เป็นคำสั่งเปรียบเทียบ
4. *Transcendental* เป็นคำสั่งคำนวณฟังก์ชั่นตรีโกณมิติ, ยกกำลัง, log
5. *Load Constants* เป็นคำสั่งใส่ค่าคงที่ให้ค่าใน Register
6. *FPU Control* เป็นคำสั่งควบคุมการทำงานของ FPU

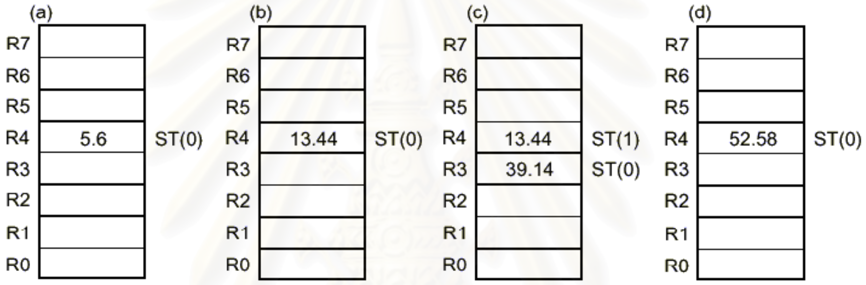
รหัสคำสั่งเครื่องที่ใช้ในการคำนวณเพื่อแก้สมการเมทริกซ์นี้จะใช้เพียงคำสั่งดังนี้

FLD	อ่านค่าตัวเลขจำนวนจริงไปเก็บใน Register ของหน่วยประมวลผลกลาง
FSTP	บันทึกค่าตัวเลขจำนวนจริงไปเก็บที่หน่วยความจำที่กำหนด
FLDZ	ใส่ค่า 0 ลงใน Register ของ หน่วยประมวลผลกลาง
FLD1	ใส่ค่า 1 ลงใน Register ของ หน่วยประมวลผลกลาง
FADDP	บวกเลขจำนวนจริง
FSUBR	ลบเลขจำนวนจริง
FMUL	คูณเลขจำนวนจริง
FDIVRP	หารเลขจำนวนจริง

เนื่องจากในงานวิจัยนี้ ข้อมูลเลขจำนวนจริงแต่ละตัวมีขนาด 8 ไบต์ (64 บิต) การใช้รหัสคำสั่งเครื่องต่างๆจึงได้เขียนในรูป

```
Fxxx    QWORD [ offset ]
```

ซึ่งหมายถึงเป็นการนำค่าเลขจำนวนจริงที่อยู่ใน Address [DS:offset] ถึง [DS:offset+7] มาคำนวณกับค่าใน Register ใน FPU โดย Register ที่ FPU ใช้คำนวณเลขจำนวนจริงจะเป็น Register ชุดอีกชุดต่างจาก Register พื้นฐานของหน่วยประมวลผลกลาง ประกอบด้วย Register ขนาด 80 บิต จำนวน 8 ตัว เรียกว่า R0 - R7 อยู่ในโครงสร้างข้อมูลแบบกองซ้อน (Stack) คำสั่งในการคำนวณเลขจำนวนจริงทั้งหมดจะกระทำกับ Register ที่อยู่ที่ในตำแหน่งบนสุดของกองซ้อน (Top of Stack) ที่เรียกว่า ST(0) หรือ ST เมื่อมีการอ่านค่าใหม่เข้ามาหรือนำผลลัพธ์จากการคำนวณมาเก็บไว้ในกองซ้อน (PUSH) จะทำให้ ST ถูกเลื่อนไปชี้ Register (R) ตัวถัดไป แต่ถ้ามีการนำค่าออก (POP) จาก Stack จะทำให้ ST ถูกเลื่อนถอยกลับมา



```
FLD      value1        ; (a) value1 = 5.6
FMUL     value2        ; (b) value2 = 2.4
FLD      value3        ; value3 = 3.8
FMUL     value4        ; (c) value4 = 10.3
FADDP    ST(1)         ; (d)
```

รูปที่ 5.7 การเก็บค่าตัวเลขจำนวนจริงของ Register ใน FPU

ตัวอย่างรหัสคำสั่งเครื่องในรูปที่ 5.7 มีขั้นตอนการทำงานดังนี้

1. คำสั่ง FLD value1 จะนำค่า 5.6 ไปเก็บที่ ST(0) ซึ่งสมมุติว่ากำลังชี้อยู่ที่ R4
2. คำสั่ง FMUL value2 ค่า 2.4 จะถูกนำไปคูณกับ ST(0) ได้ค่าเป็น 13.44 เก็บอยู่ที่ R4
3. คำสั่ง FLD value3 จะ PUSH ค่าลงในกองซ้อน ST จะเลื่อนลงมาชี้ที่ R3 แล้วจึงเก็บค่า 3.8 ลงไป
4. คำสั่ง FMUL value4 จะคูณค่า 3.8 ใน ST(0) กับค่า 10.3 ได้เป็น 39.14 เก็บใน R5
5. คำสั่ง FADDP ST(1) จะนำค่าใน ST(1) คือ 13.44 มาบวกกับค่าใน ST(0) และ POP ค่า 39.14 ออกมาด้วย ทำให้ ST(0) เลื่อนกลับมาที่ R4 แล้วจึงนำค่าผลบวกมาเก็บที่ R4

ตัวอย่างข้างบนแสดงถึงการใช้รหัสคำสั่งเครื่องเพื่อหาผลรวมของการคูณเลขจำนวนจริง ซึ่งเป็นการคำนวณขั้นตอนหนึ่งที่สำคัญในการแก้สมการเมทริกซ์ของวงจรไฟฟ้า

5.2.2 รูปแบบของรหัสคำสั่งเครื่อง

เมื่อโปรแกรมหาวิธีแก้สมการเมทริกซ์ของวงจรได้แล้ว จะสร้างชุดรหัสคำสั่งเครื่องสำหรับแก้สมการเก็บเรียงต่อกันในแถวลำดับชื่อ Mac_Code รหัสคำสั่งเครื่องนี้สามารถแบ่งออกได้เป็น 9 แบบย่อย เพื่อที่จะทำให้ชุดรหัสคำสั่งเครื่องมีความยาวน้อยที่สุด

5.2.2.1 การคูณแล้วเก็บค่าผลลัพธ์ใน ST

คือการนำค่าสมาชิกที่ตำแหน่งดัชนี a และ b มาคูณกัน แล้วเก็บค่าไว้ใน ST รูปแบบคำสั่งนี้จะใช้เป็นคำสั่งแรกในการหาค่าผลรวมของการคูณ โดยที่จะใช้ ST เป็นตัวแปรชั่วคราวเก็บค่าผลรวม สมการที่ (5.5) แสดงสูตรการคำนวณดังนี้

$$ST = A[a] * A[b] \tag{5.5}$$

รหัสคำสั่งเครื่องในแถวลำดับ Mac_Code จะมีรูปแบบตามรูปที่ 5.8

FLD	QWORD A[a]	(ST ⇦ A[a])	DD	06	Index a
FMUL	QWORD A[b]	(ST ⇦ ST*A[b])	DC	0E	Index b

รูปที่ 5.8 รูปแบบรหัสคำสั่งเครื่อง"การคูณแล้วเก็บค่าผลลัพธ์ใน ST" ในแถวลำดับ Mac_Code

5.2.2.2 การคูณแล้วเก็บค่าผลลัพธ์ในเมทริกซ์

คือการนำค่าสมาชิกที่ตำแหน่งดัชนี a คูณกับค่าใน ST แล้วเก็บค่าผลลัพธ์ในสมาชิกที่ตำแหน่งดัชนี b สมการที่ (5.5) แสดงสูตรการคำนวณดังนี้

$$A[b] = A[a] * ST \tag{5.5}$$

การเก็บรหัสคำสั่งเครื่องจะเก็บลงในแถวลำดับ Mac_Code ตามรูปแบบในรูปที่ 5.9

FMUL	QWORD A[a]	(ST ⇦ ST*A[a])	DC	0E	Index a
FSTP	QWORD A[b]	(A[b] ⇦ ST)	DD	1E	Index b

รูปที่ 5.9 รูปแบบรหัสคำสั่งเครื่อง"การคูณแล้วเก็บค่าผลลัพธ์ในเมทริกซ์" ในแถวลำดับ Mac_Code

5.2.2.3 การบวกค่าใน ST

คือการบวกของ Register ใน FPU ที่ตำแหน่ง ST และ ST(1) แล้วเก็บค่าผลลัพธ์ใน ST ดังแสดงไว้ในสมการที่ (5.7)

$$ST = ST + ST(1) \tag{5.7}$$

การเก็บรหัสคำสั่งเครื่องจะเก็บลงในแถวลำดับ Mac_Code ตามรูปแบบในรูปที่ 5.10

FADDP	ST(1)	(ST ⇦ ST*ST(1))	DE	C1

รูปที่ 5.10 รูปแบบรหัสคำสั่งเครื่อง "การบวกค่าใน ST" ในแถวลำดับ Mac_Code

5.2.2.4 การลบค่าด้วย ST

คือการนำค่าสมาชิกที่ตำแหน่งดัชนี a มาลบออกจากค่าใน ST และเก็บค่าผลลัพธ์ใน ST ดังแสดงไว้ในสมการที่ (5.8)

$$ST = A[a] - ST \tag{5.8}$$

การเก็บรหัสคำสั่งเครื่องจะเก็บลงในแถวลำดับ Mac_Code ตามรูปแบบในรูปที่ 5.11

FSUBR	QWORD A[a]	(ST ⇦ A[b] - ST)	DC	2E	Index a

รูปที่ 5.11 รูปแบบรหัสคำสั่งเครื่อง”การลบค่าด้วย ST” ในแถวลำดับ Mac_Code

5.2.2.5 การเก็บค่าลงใน ST

ใช้ในกรณีที่สมาชิกที่ตำแหน่งดัชนี a เป็นสมาชิกในแนวทแยงมุขสำคัญที่จะถูกกลับเศษส่วนในรูปแบบรหัสคำสั่งเครื่องลำดับถัดไป แต่ค่าสมาชิกนี้ยังไม่ได้ถูกเก็บใน ST จึงต้องเก็บค่าลงใน ST ก่อนการกลับเศษส่วน ดังแสดงไว้ในสมการที่ (5.9)

$$ST = A[a] \tag{5.9}$$

การเก็บรหัสคำสั่งเครื่องจะเก็บลงในแถวลำดับ Mac_Code ตามรูปแบบในรูปที่ 5.12

FLD	QWORD A[a]	(ST ⇦ A[b])	DD	06	Index a

รูปที่ 5.12 รูปแบบรหัสคำสั่งเครื่อง”การเก็บค่าลงใน ST” ในแถวลำดับ Mac_Code

5.2.2.6 การกลับเศษส่วน

คือการกลับเศษส่วนค่าใน ST ตามเทคนิคการกลับเศษส่วนค่าสมาชิกในแนวทแยงมุขสำคัญ (หัวข้อ 4.4) ดังแสดงไว้ในสมการที่ (5.10)

$$ST = 1 / ST \tag{5.10}$$

การเก็บรหัสคำสั่งเครื่องจะเก็บลงในแถวลำดับ Mac_Code ตามรูปแบบในรูปที่ 5.13

FLD1		(ST ⇦ 1)	D9	E8
FDIVRP	ST(1)	(ST ⇦ ST / ST(1))	DE	F1

รูปที่ 5.13 รูปแบบรหัสคำสั่งเครื่อง”การกลับเศษส่วน” ในแถวลำดับ Mac_Code

5.2.2.7 การบันทึกค่า ST ลงในเมทริกซ์

คือการเก็บค่าใน ST ลงในสมาชิกที่ตำแหน่งดัชนี a ดังแสดงไว้ในสมการที่ (5.11)

$$A[a] = ST \tag{5.11}$$

การเก็บรหัสคำสั่งเครื่องจะเก็บลงในแถวลำดับ Mac_Code ตามรูปแบบในรูปที่ 5.14

FSTP	QWORD A[a]	(A[b] ⇦ ST)	DD	1E	Index a

รูปที่ 5.14 รูปแบบรหัสคำสั่งเครื่อง”การบันทึกค่า ST ลงในเมทริกซ์” ในแถวลำดับ Mac_Code

5.2.2.8 กำหนดค่าเป็นศูนย์

คือการกำหนดค่าสมาชิกที่ตำแหน่งดัชนี a ให้มีค่าเป็นศูนย์ จะใช้เป็นคำสั่งแรกสุดเพื่อกำหนดค่าสมาชิกที่เป็น Fill-ins ใหม่ให้มีค่าเป็นศูนย์ก่อนเริ่มการคำนวณ การเก็บรหัสคำสั่งเครื่องจะเก็บลงในแถวลำดับ Mac_Code ตามรูปแบบในรูปที่ 5.15

FLDZ		(ST ⇦ 0)	D9	EE	
FSTP	QWORD A[a]	(A[b] ⇦ ST)	DD	1E	Index a

รูปที่ 5.15 รูปแบบรหัสคำสั่งเครื่อง”กำหนดค่าเป็นศูนย์” ในแถวลำดับ Mac_Code

5.2.2.9 หยุดการคำนวณ

คือการสั่งให้เครื่องออกจากการคำนวณด้วยรหัสคำสั่งเครื่องที่สร้างขึ้น จะใช้เป็นคำสั่งสุดท้ายของรหัสคำสั่งเครื่องเสมอ โดยจะเก็บลงในแถวลำดับ Mac_Code ตามรูปแบบในรูปที่ 5.16

RETF	(Return far)	CB

รูปที่ 5.16 รูปแบบรหัสคำสั่งเครื่อง”หยุดการคำนวณ” ในแถวลำดับ Mac_Code

จากรูปแบบรหัสคำสั่งเครื่องทั้ง 8 แบบจะสามารถนำมาเรียงต่อกันเป็นชุดรหัสคำสั่งเครื่องเพื่อแก้สมการวงจรไฟฟ้าได้ เช่นในการคำนวณหาค่า l_{ik} ตามสมการ (2.6)

$$l_{ik} = a_{ik} - \sum_{m=1}^{k-1} l_{im} u_{mk} \qquad \text{โดย } i \geq k \tag{2.6}$$

ชุดรหัสคำสั่งเครื่องจะมีลักษณะดังนี้

```
FLD     QWORD [$l_{i1}$]        ; รูปแบบที่ 1
FMUL    QWORD [$u_{1k}$]
FLD     QWORD [$l_{i2}$]        ; รูปแบบที่ 1
FMUL    QWORD [$u_{2k}$]
FADDP   ST(1)                  ; รูปแบบที่ 3
…
FLD     QWORD [$l_{i\,k-1}$]     ; รูปแบบที่ 1
FMUL    QWORD [$u_{k-1\,k}$]
```

FADDP ST(1) ; รูปแบบที่ 3

FSUBR QWORD [a_{ik}] ; รูปแบบที่ 4

ซึ่งจะได้ค่าผลลัพธ์ของ l_{ik} เก็บอยู่ใน ST ถ้า $i = k$ แสดงว่า l_{ik} เป็นสมาชิกในแนวทแยงมุขสำคัญของเมทริกซ์ A จึงต้องกลับเศษส่วนตามเทคนิคการกลับเศษส่วนค่าสมาชิกในแนวทแยงมุขสำคัญ แล้วเขียนค่ากลับไปที่ l_{ik} ดังนี้

FLD1 ; รูปแบบที่ 6

FDIVRP ST(1)

FSTP QWORD [l_{ik}] ; รูปแบบที่ 7

ส่วนการคำนวณหาค่า u_{ki}, z_i และ x_i จากสูตร (2.7), (2.9) และ (2.10) ตามลำดับ จะมีลักษณะการคำนวณและรหัสคำสั่งเครื่องที่ใช้ในการหาค่าคล้ายคลึงกับการหาค่า l_{ik} และชุดรหัสคำสั่งเครื่องที่เป็นสูตรสำเร็จการแก้สมการเมทริกซ์จะเป็นรหัสคำสั่งเครื่องสำหรับคำนวณค่า l_{ik}, u_{ki}, z_i และ x_i เรียงต่อกันตามขั้นตอน LU Factorization แล้วต่อท้ายด้วยคำสั่ง RETF (Return Far) เป็นการสิ้นสุดการคำนวณด้วยสูตรสำเร็จที่เป็นรหัสคำสั่งเครื่อง

แถวลำดับ Mac_Code สามารถเปลี่ยนแปลงขนาดความยาวได้เช่นเดียวกับแถวลำดับ Inp_Code แต่เริ่มแรกจะกำหนดความยาวของแถวลำดับไว้ 6 x n^2 ไบต์ (n คือจำนวนตัวแปรอิสระในสมการวงจร) แต่ถ้าหากรหัสคำสั่งเครื่องมีขนาดใหญ่กว่านี้ โปรแกรมจะจองแถวลำดับ Mac_Code ใหม่ที่มีขนาดใหญ่ขึ้นแล้วย้ายรหัสคำสั่งเครื่องมาเก็บไว้ที่แถวลำดับใหม่นี้ และคืนพื้นที่หน่วยความจำของ Mac_Code เดิมกลับให้กับระบบ

5.2.3 วิธีการเรียกใช้รหัสคำสั่งเครื่อง

เมื่อสร้างรหัสคำสั่งเครื่องไว้ในแถวลำดับ Mac_Code แล้ว จะสั่งให้คอมพิวเตอร์ทำงานตามรหัสคำสั่งเครื่องที่สร้างขึ้นโดยการกำหนดให้ Register CS:IP ชี้ไปยังแถวลำดับ Mac_Code วิธีการคือเรียกใช้คำสั่ง CALL FAR ตามขั้นตอนดังนี้

```
PUSH      ES                  ; เก็บ Extra Segment (ES) เดิมไว้ใน Stack
PUSH      DS                  ; เก็บ Data Segment (DS) เดิมไว้ใน Stack
MOV       ES, [Segment of b]  ; ให้ ES ชี้ไปยัง Segment ของแถวลำดับ b
MOV       DS, [Segment of A]  ; ให้ DS ชี้ไปยัง Segment ของแถวลำดับ A
CALL FAR  [StartInst]         ; กระโดดไปทำงานที่แถวลำดับ StartInst
POP       DS                  ; เมื่อทำงานเสร็จแล้วจึงคืนค่า DS เดิม
POP       ES                  ; และค่า ES เดิม
```

StartInst เป็นแถวลำดับที่เก็บคำสั่ง CALL FAR ให้หน่วยประมวลผลการกลางกระโดดไปทำงานยังแถวลำดับ Mac_Code อีกทอดหนึ่ง ทั้งนี้เนื่องจากในโปรแกรมสามารถสร้าง Mac_Code

ขึ้นมาได้หลายชุด จึงไม่มีตำแหน่ง Address แน่นอนที่จะกำหนดลงในโปรแกรมได้ ดังนั้นจึงต้องสร้างแถวลำดับ StartInst ขึ้นมาโดยเก็บข้อมูลตามรูปที่ 5.17 วิธีนี้จึงสามารถสั่งให้คอมพิวเตอร์ไปทำงานที่ Address ใดๆได้ โดยกำหนดค่า Address นั้นไว้ในแถวลำดับ StartInst ก่อน

CALL FAR [Mac_Code]	9A	Address ของ Mac_Code

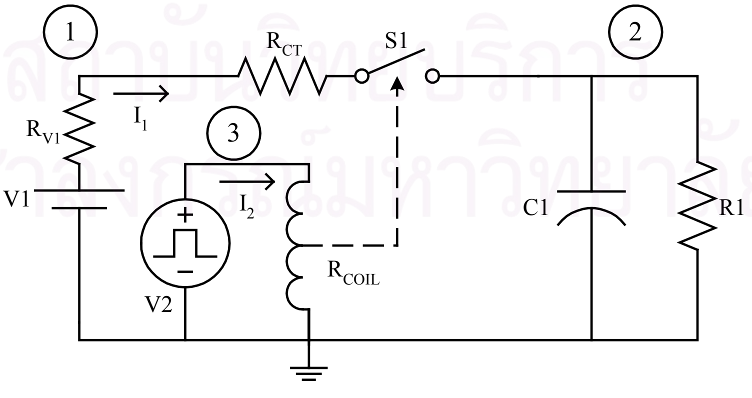
รูปที่ 5.17 ข้อมูลที่เก็บในแถวลำดับ StartInst

ส่วนการกำหนดค่า Register DS และ ES เป็นค่า Segment ของเมทริกซ์ A และ b ตามลำดับนั้นเพราะในการคำนวณจะต้องอ่านค่าจากทั้งเมทริกซ์ A และ b ซึ่งไม่ได้อยู่ที่ Segment เดียวกัน แต่โดยปรกติแล้วรหัสคำสั่งเครื่องจะอ้างถึงได้เพียงข้อมูลใน Segment DS เท่านั้น ดังนั้นถ้าต้องการใช้ข้อมูลจากเมทริกซ์ b จะต้องใส่รหัส 26h ไว้นำหน้ารหัสคำสั่งเพื่อบอกว่าจะอ้างถึงข้อมูลใน Segment ES การที่กำหนดให้ ES เก็บ Segment ของเมทริกซ์ b เพราะมีการใช้ข้อมูลจากเมทริกซ์ b น้อยกว่า ซึ่งจะทำให้ใช้เนื้อที่เก็บรหัส 26h น้อยกว่า

เนื่องจากการคำนวณโดยใช้รหัสคำสั่งเครื่องมีการใช้เทคนิคการเรียงลำดับใหม่เช่นเดียวกับการคำนวณโดยใช้รหัสแบบแปลย่อย ดังนั้นเมื่อคำนวณตามรหัสคำสั่งเครื่องเสร็จเรียบร้อยแล้วจะต้องสลับค่าในเมทริกซ์ b ให้ถูกต้องก่อนจะนำคำตอบไปแสดงผลต่อไป

5.3 ปัญหาในการใช้สูตรสำเร็จการแก้สมการเมทริกซ์

การแก้สมการเมทริกซ์โดยใช้สูตรสำเร็จมีปัญหาสำคัญที่ต้องคำนึงถึง คือปัญหาที่ Pivot ตอนเริ่มแรกมีค่าไม่เป็นศูนย์ แต่เมื่อเวลาผ่านไปเกิดการเปลี่ยนสถานะทำให้ Pivot กลายเป็นศูนย์ไป ทำให้เกิดข้อผิดพลาด division by zero ขึ้น ปัญหานี้จะเกิดขึ้นได้ในการจำลองหาผลตอบสนองชั่วขณะของวงจรไฟฟ้าบางวงจรที่มีอุปกรณ์ประเภทสวิทช์เป็นองค์ประกอบ เช่น วงจรตัวอย่างในรูปที่ 5.18



V2	เป็นแหล่งแรงดันพัลซ	R_{CT}	คือความต้านทานที่หน้าสัมผัสของสวิทช์ S1
R_{V1}	คือความต้านทานภายในแหล่งแรงดัน V1	R_{COIL}	คือความต้านทานของขดลวดควบคุมสวิทช์ S1

รูปที่ 5.18 วงจรตัวอย่างที่มีสวิทช์เป็นองค์ประกอบ

ตามวงจรในรูปที่ 5.18 สวิทซ์ S1 จะปิดเมื่อแรงดันปมที่ 3 (V_3) มีค่ามากกว่า 0 โวลต์ ซึ่งเป็นเวลาที่แหล่งแรงดันพัลซ V2 จ่ายแรงดัน V2 โวลต์ ณ เวลานี้สมการวงจรจะเป็น

$$\begin{bmatrix} \frac{1}{R_{CT}} & -\frac{1}{R_{CT}} & & -1 & \\ -\frac{1}{R_{CT}} & \frac{1}{R_{CT}}+\frac{1}{R1}+\frac{C1}{h} & & & \\ & & \frac{1}{R_{COIL}} & & -1 \\ 1 & & & R_{V1} & \\ & & 1 & & \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ I_1 \\ I_2 \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{C1}{h}V_2(t-1) \\ 0 \\ V1 \\ V2 \end{bmatrix}$$

เมื่อทำการเรียงลำดับใหม่ตามวิธี Markowitz (หัวข้อ 4.3) แล้วจะได้

$$\begin{bmatrix} 1 & & & & \\ \frac{1}{R_{COIL}} & -1 & & & \\ & & \frac{1}{R_{CT}}+\frac{1}{R1}+\frac{C1}{h} & -\frac{1}{R_{CT}} & \\ & & -\frac{1}{R_{CT}} & \frac{1}{R_{CT}} & -1 \\ & & & 1 & R_{V1} \end{bmatrix} \begin{bmatrix} V_3 \\ I_2 \\ V_2 \\ V_1 \\ I_1 \end{bmatrix} = \begin{bmatrix} V2 \\ 0 \\ \frac{C1}{h}V_2(t-1) \\ 0 \\ V1 \end{bmatrix}$$

แต่ ณ เวลาที่ V2 = 0 โวลต์ จะทำให้สวิทซ์ S1 เปิด และ ค่า $\frac{1}{R_{CT}}$ เปลี่ยนเป็น 0 โอห์ม

$$\begin{bmatrix} 1 & & & & \\ \frac{1}{R_{COIL}} & -1 & & & \\ & & \frac{1}{R1}+\frac{C1}{h} & & \\ & & & 0 & -1 \\ & & & 1 & R_{V1} \end{bmatrix} \begin{bmatrix} V_3 \\ I_2 \\ V_2 \\ V_1 \\ I_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \frac{C1}{h}V_2(t-1) \\ 0 \\ V1 \end{bmatrix}$$

จะเห็นว่า PIVOT ตัวที่ 4 เปลี่ยนค่าเป็นศูนย์เมื่อเวลาผ่านไป ซึ่งถ้าหากยังแก้สมการด้วยสูตรสำเร็จการแก้สมการเมทริกซ์เดิมต่อไปจะทำให้เกิดข้อผิดพลาด divide by zero ขึ้นในขณะที่กลับเศษส่วนของ Pivot ตัวที่ 4

สำหรับสูตรสำเร็จการแก้สมการเมทริกซ์ที่เป็นรหัสแบบแปลคำสั่งสามารถป้องกันปัญหานี้ได้โดยง่าย เพียงเพิ่มคำสั่งตรวจสอบค่าของ Pivot ก่อนที่จะทำรหัสการหาร (รหัสแบบที่ 2) หากพบว่า

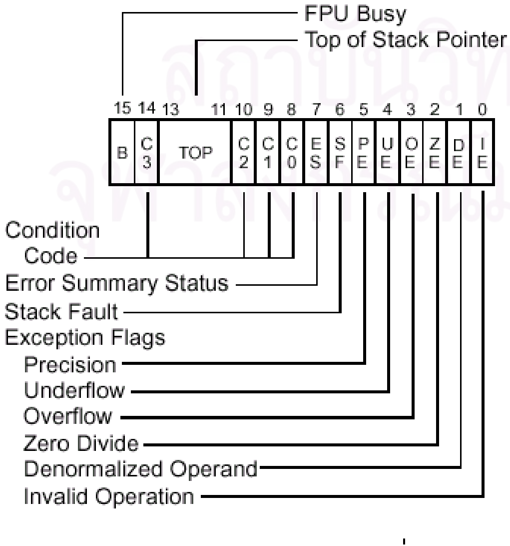
Pivot เป็นศูนย์จะออกจากการคำนวณทันทีและจะทำการเรียงลำดับใหม่แล้วจึงแยกตัวประกอบแบบแอลยูและสร้างสูตรสำเร็จการแก้สมการเมทริกซ์ใหม่เพื่อคำนวณหาคำตอบต่อไป

แต่ถ้าสูตรสำเร็จการแก้สมการเมทริกซ์เป็นรหัสคำสั่งเครื่อง จะมีวิธีตรวจสอบค่าของ Pivot อยู่ 2 วิธีคือ

1. กำหนดค่า Exception Marks สำหรับ FPU Control Word[15] ให้ดักความผิดพลาด division by zero ไว้ และเมื่อเกิดความผิดพลาดนี้ขึ้น CPU จะกระโดดไปทำงานที่ Interrupt เบอร์ 75h โดยยังไม่ออกจากการทำงานของโปรแกรม เราสามารถเปลี่ยนการทำงานของ Interrupt เบอร์ 75h ให้โปรแกรมกลับไปทำการเรียงลำดับใหม่และสร้างสูตรสำเร็จการแก้สมการใหม่ที่เลี่ยงการหารด้วยค่าศูนย์ได้ แต่วิธีนี้มีข้อเสียที่สำคัญคือทำให้การคำนวณช้าลงมาก เนื่องจากมีการตรวจสอบความผิดพลาดนี้ทุกครั้งที่คำนวณโดย FPU แม้ว่าจะไม่ใช่คำสั่ง FDIV ดังนั้นจึงไม่เลือกใช้วิธีนี้

2. เพิ่มรหัสคำสั่งเครื่องสำหรับตรวจสอบค่า Pivot ว่าเป็นศูนย์หรือไม่ ก่อนทำงานตามรหัสคำสั่งเครื่องรูปแแบบที่ 6 “การกลับเศษส่วน” โดยเรียกคำสั่ง FTST เพื่อตรวจสอบค่า Pivot ที่เก็บใน ST ซึ่งถ้า ST มีค่าเป็นศูนย์จะทำให้ C_3 Flag ใน FPU Status Word[15] (รูปที่ 5.19) ถูก set เป็น 1 ส่วนรหัสคำสั่งเครื่องที่เพิ่มเข้ามามีดังนี้

```
          FTST                          ; เปรียบเทียบ ST(0) กับค่า 0.0
          FNSTSW   AX                   ; อ่านค่า FPU Status Word มาเก็บไว้ใน AX
          AND      AH, 40h              ; กรองเฉพาะบิตที่ 14 ซึ่งเป็น Zero flag
          JZ       DIV_OP               ; ถ้า ST ≠ 0 ให้ไปทำการกลับเศษส่วน
          RETF                          ; ถ้า ST = 0 ให้ออกจากการคำนวณทันที
DIV_OP :  FLD1                          ; รูปแบบที่ 6 “การกลับเศษส่วน”
          FDIVRP   ST(1)
```

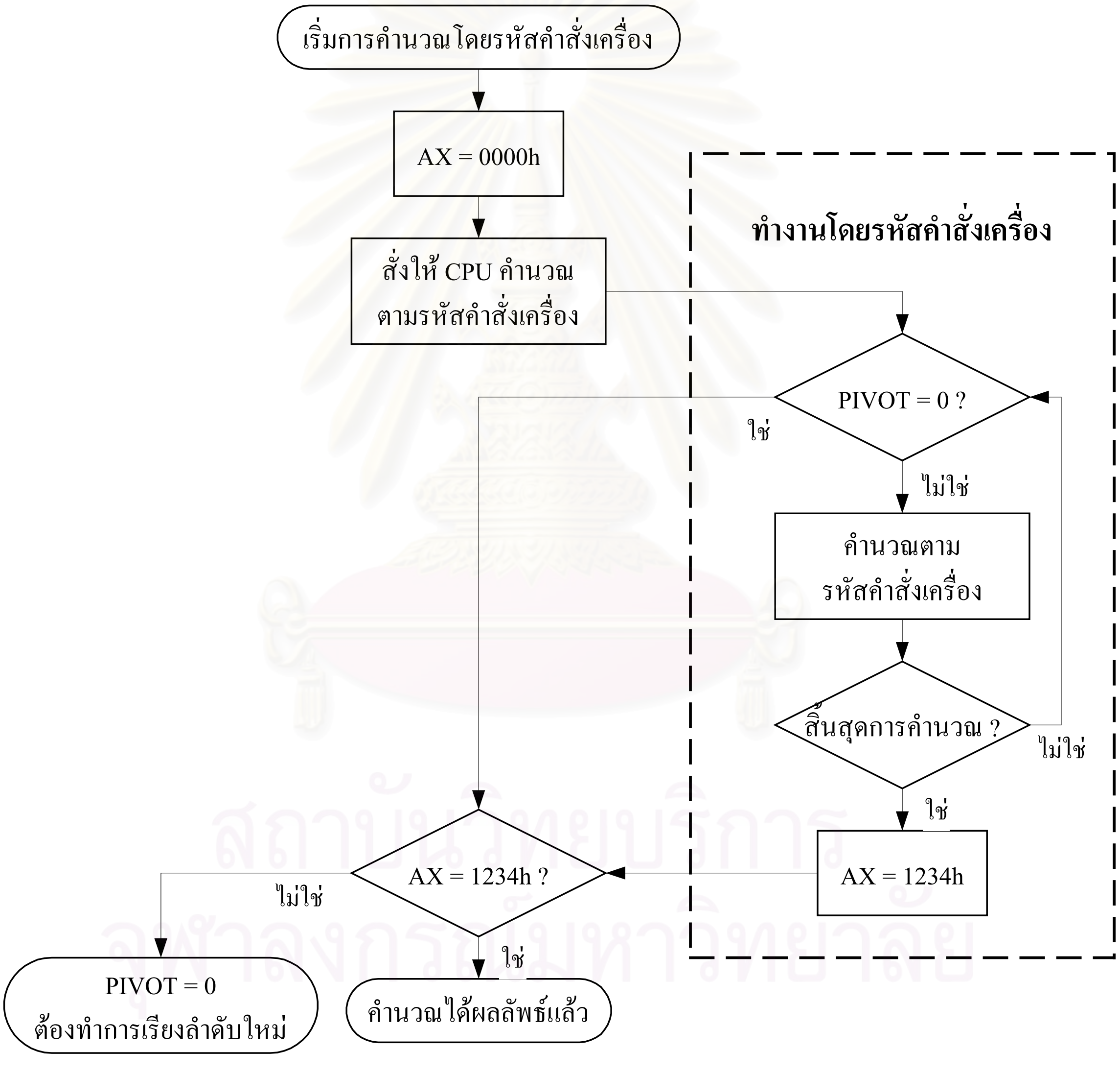


COND.CODE	FLAG
C_0	Overflow flag
C_1	(NONE)
C_2	Parity flag
C_3	Zero flag

รูปที่ 5.19 FPU Status Word

การตรวจสอบ Pivot โดยวิธีที่ 2 จะทำให้โปรแกรมใช้เวลาคำนวณเพิ่มขึ้น n*12 สัญญาณนาฬิกา (n คือจำนวนตัวแปรอิสระในวงจร) ซึ่งเทียบเท่ากับเวลาในการคูณเลขจำนวนจริง n*4 ครั้ง นอกจากนี้ยังทำให้ขนาดหน่วยความจำของรหัสคำสั่งเครื่องใหญ่ขึ้น 10 ไบต์ต่อการใช้คำสั่งหาร 1 ครั้ง ผลเสียทั้งสองด้านถือว่ายอมรับได้สำหรับการแก้ปัญหาข้างต้นเพื่อเพิ่มความเสถียรให้กับโปรแกรม จึงเลือกใช้วิธีนี้สำหรับการตรวจสอบ Pivot ก่อนการหาร

และเพื่อตรวจสอบว่าการออกจากการคำนวณโดยรหัสคำสั่งเครื่องเป็นเพราะ Pivot มีค่าเป็นศูนย์ หรือเป็นเพราะสิ้นสุดการคำนวณตามปรกติ จึงต้องใช้ Register AX เป็นตัวชี้ ตามแผนภูมิสายงานดังรูปที่ 5.20



รูปที่ 5.20 แผนภูมิสายงานการคำนวณโดยรหัสคำสั่งเครื่อง

บทที่ 6

ทดสอบและวิจารณ์ผล

บทนี้จะกล่าวถึงผลการทดสอบประสิทธิภาพของเทคนิคเมทริกซ์ของมากเลขศูนย์ โดยการเปรียบเทียบขนาดหน่วยความจำและเวลาที่ใช้ในการจำลองการทำงานของวงจรไฟฟ้าระหว่างโปรแกรม “เล็ก 6.0” และโปรแกรม “เล็ก” ที่ได้ปรับปรุงเพิ่มเติมเทคนิคของเมทริกซ์มากเลขศูนย์แล้ว และท้ายสุดจะเป็นการวิเคราะห์ผลการทดสอบโดยนำเสนอทั้งจุดเด่นและจุดด้อยของเทคนิคของเมทริกซ์มากเลขศูนย์

6.1 <u>รายละเอียดของเครื่องคอมพิวเตอร์ที่ใช้ในการทดสอบ</u>

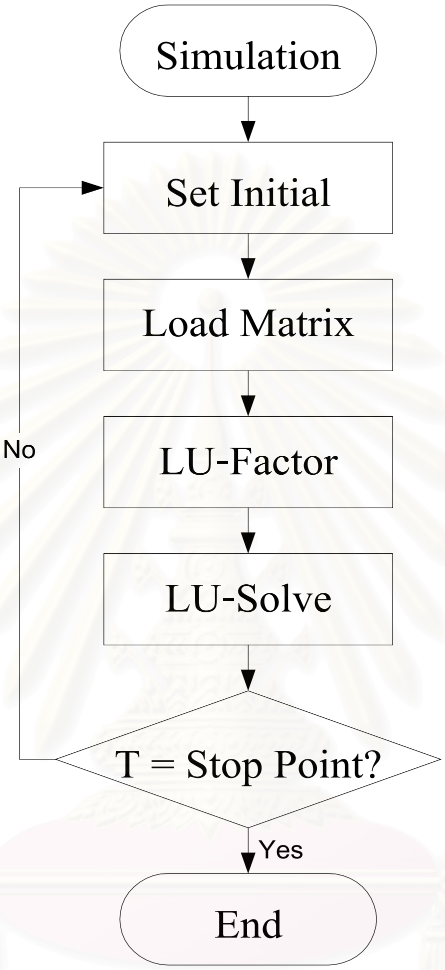
1. หน่วยประมวลผลกลาง (CPU)	Pentium 100 MHz
2. หน่วยความจำชั่วคราว (RAM)	16 Mbytes
3. หน่วยความจำแคช (Cache)	256 Kbytes
4. ระบบปฏิบัติการ	MS-DOS 6.2

6.2 <u>วงจรไฟฟ้าที่ใช้ในการทดสอบ</u>

วงจรไฟฟ้าที่เลือกใช้ในการทดสอบมีทั้งหมด 11 วงจร โดย 5 วงจรแรกเป็นวงจรที่มีใช้จริงในทางปฏิบัติได้แก่วงจร Regulator, วงจร Buck, วงจร OR Gate, วงจร Phase-Splitting และ วงจร Triangular Wave Generator ส่วนอีก 6 วงจรเป็นวงจร RC Ladder ที่มีจำนวนปมในวงจรต่างๆกัน โดยได้แสดงรายละเอียดของแต่ละวงจรในภาคผนวก ก.

6.3 โปรแกรมที่นำมาทดสอบ

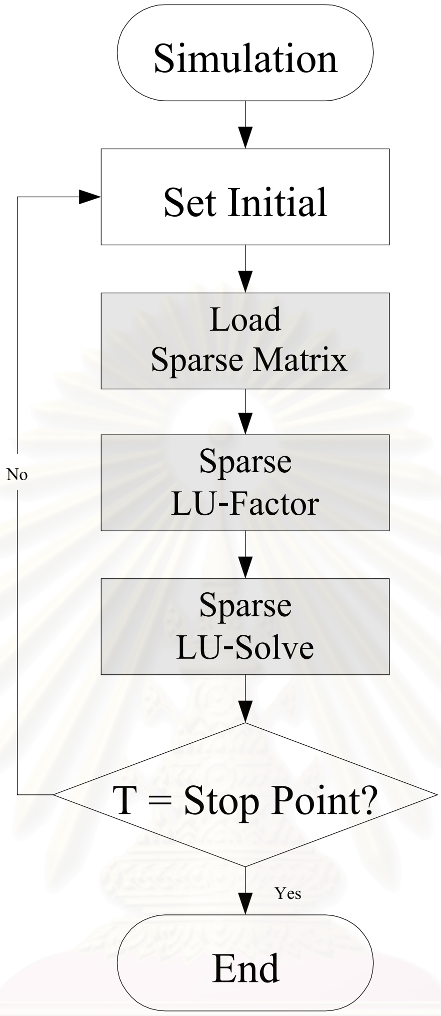
6.3.1 โปรแกรม “เล็ก 6.0”



รูปที่ 6.1 แผนภูมิการทำงานอย่างง่ายของโปรแกรม “เล็ก 6.0”

โปรแกรม “เล็ก 6.0” เป็นโปรแกรมต้นแบบที่ใช้ในงานวิทยานิพนธ์นี้ เป็นโปรแกรมที่ยังไม่ใช้เทคนิคของเมทริกซ์มากเลขศูนย์ช่วยในการคำนวณ รายละเอียดของโปรแกรม “เล็ก 6.0” ได้เสนอไว้ในบทที่ 3 แล้ว โดยรูปที่ 6.1 แสดงแผนภูมิการทำงานอย่างง่ายของโปรแกรม “เล็ก 6.0”

6.3.2 **โปรแกรม** “เล็ก 6S”



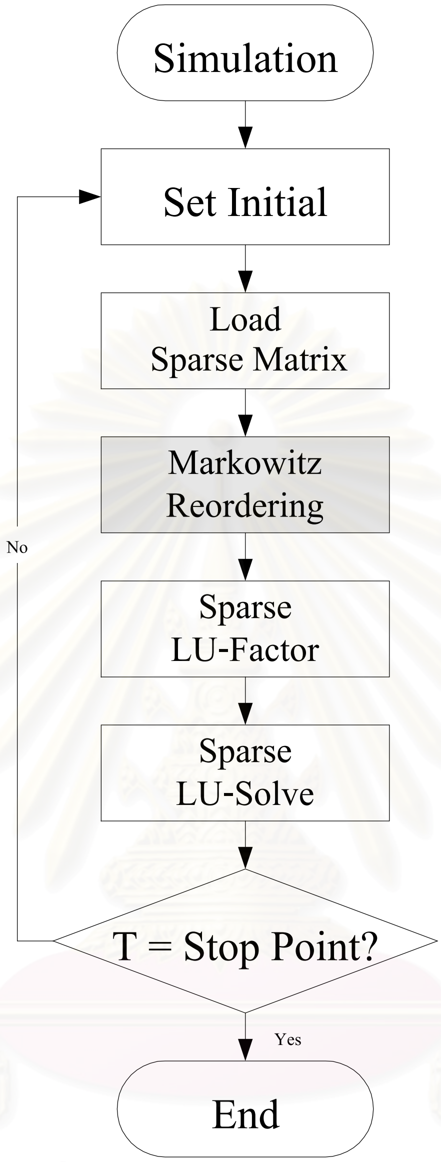
รูปที่ 6.2 แผนภูมิการทำงานอย่างง่ายของโปรแกรม “เล็ก 6S”

โปรแกรม “เล็ก 6S” เป็นโปรแกรมที่พัฒนามาจากโปรแกรม “เล็ก 6.0” โดยได้เพิ่มเทคนิคของเมทริกซ์มากเลขศูนย์ที่ได้เสนอไว้ในบทที่ 4 บางส่วนเข้าไป ได้แก่

1. การข้ามการคำนวณกับค่าศูนย์
2. โครงสร้างการเก็บข้อมูลแบบเมทริกซ์มากเลขศูนย์

ส่วนของโปรแกรม “เล็ก 6S” ที่ต่างไปจากโปรแกรม “เล็ก 6.0” คือส่วน Load Sparse Matrix, Sparse LU-Factor, Sparse LU-Solve ดังแสดงในรูปที่ 6.2 ผลการทดสอบโปรแกรม “เล็ก 6.0” กับ “เล็ก 6S” จะทำให้ทราบถึงประสิทธิภาพของเทคนิคการข้ามการคำนวณกับค่าศูนย์กับการใช้โครงสร้างการเก็บข้อมูบแบบเมทริกซ์มากเลขศูนย์

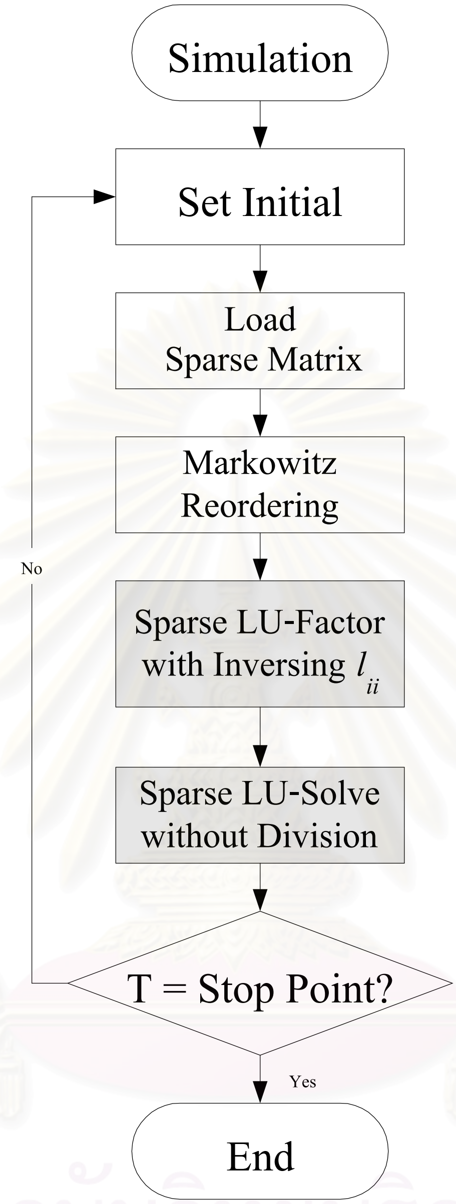
6.3.3 **โปรแกรม “เล็ก 6R”**



รูปที่ 6.3 แผนภูมิการทำงานอย่างง่ายของโปรแกรม “เล็ก 6R”

โปรแกรม “เล็ก 6R” ได้พัฒนาต่อจากโปรแกรม “เล็ก 6S” โดยเพิ่มเทคนิคการเรียงลำดับใหม่ตามวิธีของ Markowitz เข้าไป โดยจะทำขั้นตอนการเรียงลำดับใหม่ก่อนขั้นตอนการแยกตัวประกอบแอล-ยู ดังแสดงขั้นตอนการทำงานในรูปที่ 6.3 ผลการทดสอบโปรแกรม “เล็ก 6S” กับ “เล็ก 6R” จะทำให้ทราบถึงประสิทธิภาพของเทคนิคการเรียงลำดับใหม่ตามวิธีของ Markowitz

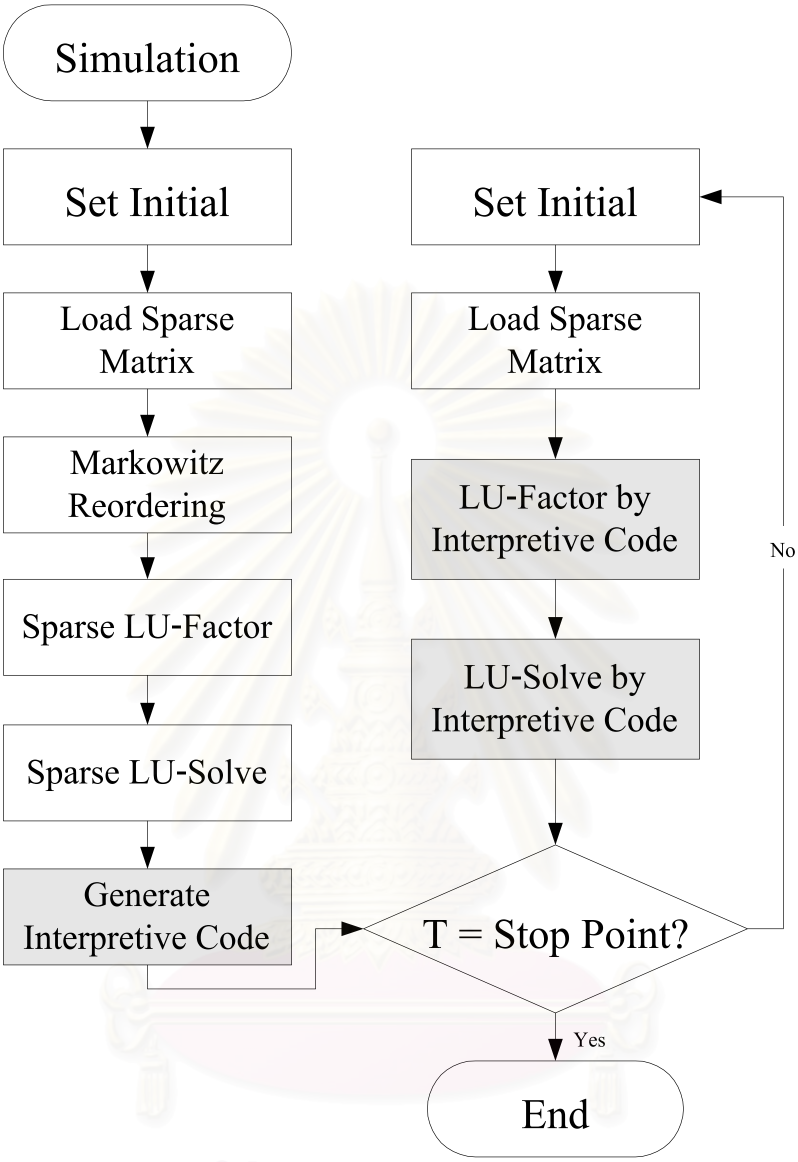
6.3.4 **โปรแกรม** “เล็ก 6RInv”



รูปที่ 6.4 แผนภูมิการทำงานอย่างง่ายของโปรแกรม “เล็ก 6RInv”

โปรแกรม “เล็ก 6RInv” ได้เพิ่มเทคนิคการกลับเศษส่วนค่าสมาชิกในแนวทแยงมุขสำคัญในขั้นตอนแยกตัวประกอบแอล-ยู ซึ่งทำให้ลดจำนวนการหารในโปรแกรมลงได้ ขั้นตอนการทำงานของโปรแกรม “เล็ก 6RInv” แสดงดังรูปที่ 6.4 ผลการทดสอบโปรแกรม “เล็ก 6R” กับ “เล็ก 6RInv” จะทำให้ทราบถึงประสิทธิภาพของเทคนิคการกลับเศษส่วนค่าสมาชิกในแนวทแยงมุขสำคัญ

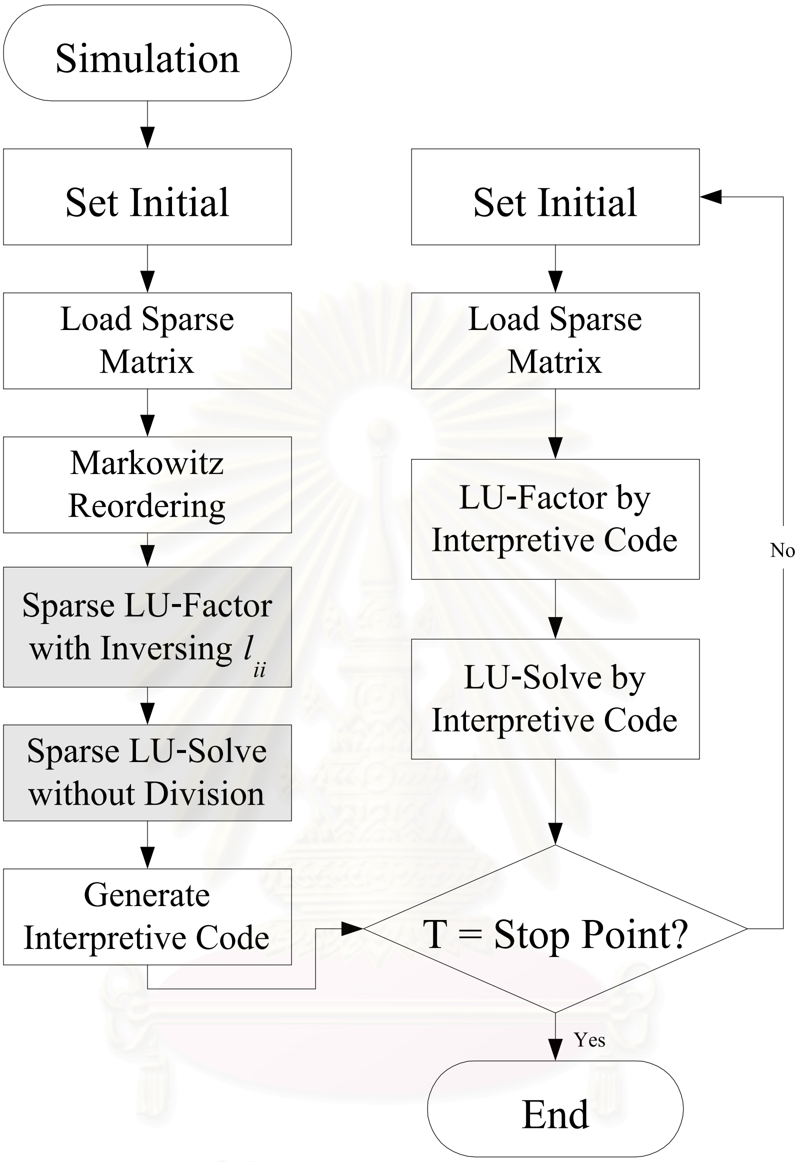
6.3.5 **โปรแกรม “เล็ก 6I”**



รูปที่ 6.5 แผนภูมิการทำงานอย่างง่ายของโปรแกรม “เล็ก 6I”

โปรแกรม “เล็ก 6I” เป็นโปรแกรมที่พัฒนาต่อจากโปรแกรม “เล็ก 6R” โดยนอกจากจะใช้เทคนิคของเมทริกซ์มากเลขศูนย์แล้ว ยังใช้เทคนิคการสร้างสูตรสำเร็จการแก้สมการเมทริกซ์ในรูปรหัสแบบแปลคำสั่งอีกด้วย โดยรายละเอียดของรหัสแบบแปลคำสั่งได้เสนอไว้ในหัวข้อที่ 5.1 ในบทที่ 5 แล้ว โดยรูปที่ 6.5 แสดงแผนภูมิการทำงานอย่างง่ายของโปรแกรม “เล็ก 6I” ผลการทดสอบโปรแกรม “เล็ก 6R” กับ “เล็ก 6I” จะทำให้ทราบถึงประสิทธิภาพของเทคนิคการสร้างสูตรสำเร็จการแก้สมการเมทริกซ์ในรูปรหัสแบบแปลคำสั่ง

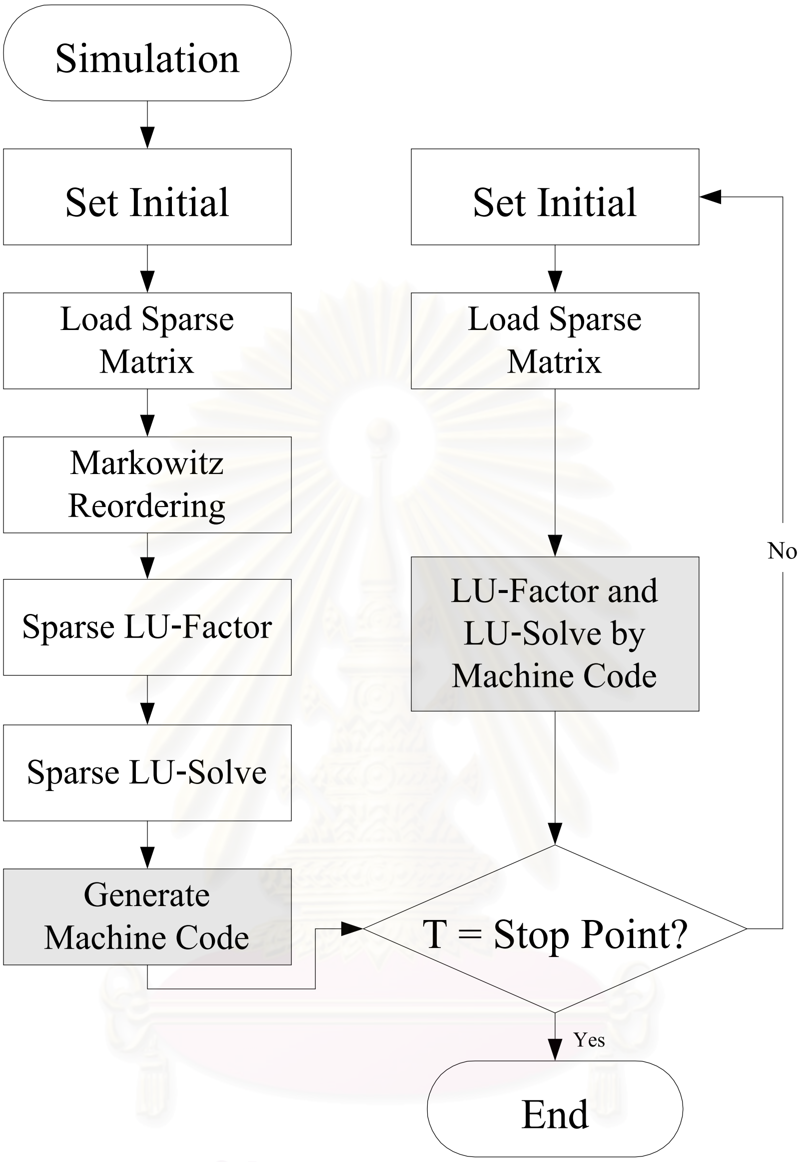
6.3.6 **โปรแกรม** “เล็ก 6IInv”



รูปที่ 6.6 แผนภูมิการทำงานอย่างง่ายของโปรแกรม “เล็ก 6IInv”

โปรแกรม “เล็ก 6IInv” พัฒนาต่อจากโปรแกรม “เล็ก 6I” โดยเพิ่มเทคนิคการกลับเศษส่วนค่าสมาชิกในแนวทแยงมุขสำคัญ เพื่อทดสอบผลของเทคนิคนี้เมื่อใช้รวมกับเทคนิคการสร้างสูตรสำเร็จการแก้สมการเมทริกซ์ในรูปรหัสแบบแปลคำสั่ง โดยรูปที่ 6.6 แสดงแผนภูมิการทำงานอย่างง่ายของโปรแกรม “เล็ก 6IInv”

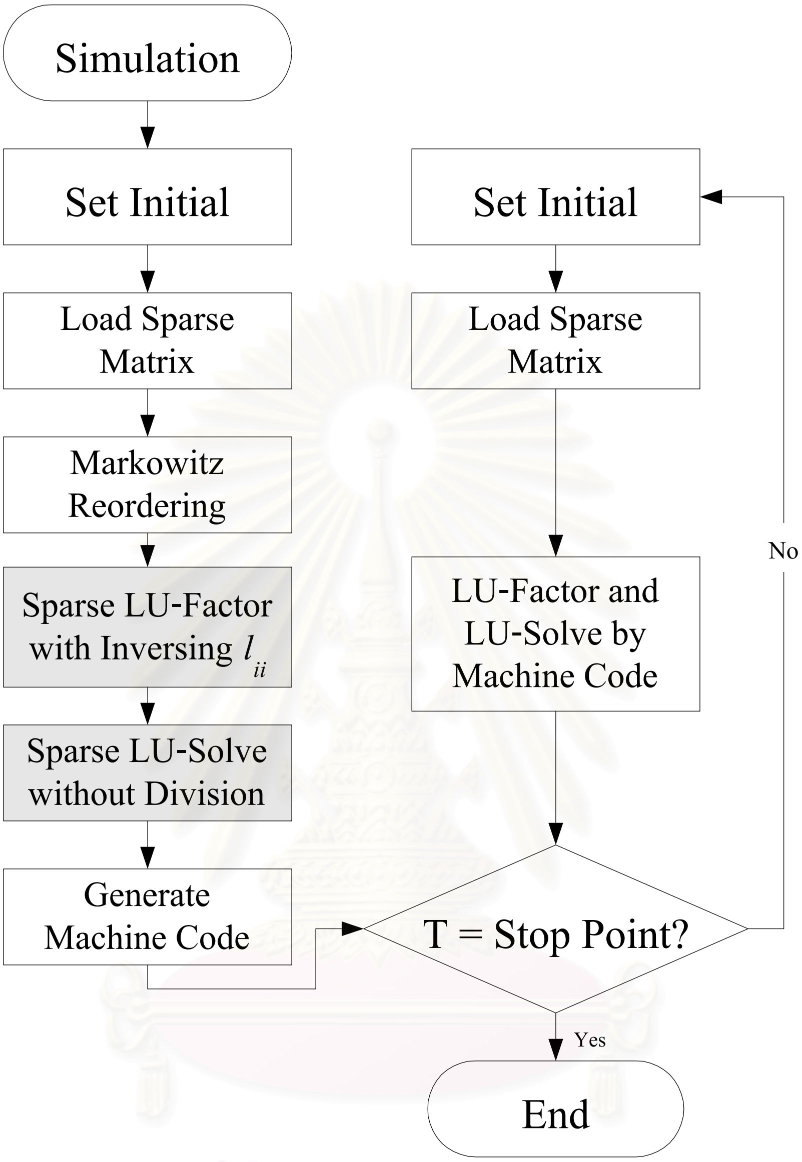
6.3.7 โปรแกรม “เล็ก 6M”



รูปที่ 6.7 แผนภูมิการทำงานอย่างง่ายของโปรแกรม “เล็ก 6M”

โปรแกรม “เล็ก 6M” เป็นโปรแกรมที่พัฒนาต่อจากโปรแกรม “เล็ก 6R” คล้ายกับโปรแกรม “เล็ก 6I” แต่ต่างกันที่ใช้เทคนิคการสร้างสูตรสำเร็จการแก้สมการเมทริกซ์ในรูปรหัสคำสั่งเครื่องแทนที่ใช้รหัสแบบแปลคำสั่ง โดยรายละเอียดของรหัสคำสั่งเครื่องได้เสนอไว้ในหัวข้อที่ 5.2 ในบทที่ 5 แล้ว โดยรูปที่ 6.7 แสดงแผนภูมิการทำงานอย่างง่ายของโปรแกรม “เล็ก 6M” ผลการทดสอบโปรแกรม “เล็ก 6R” กับ “เล็ก 6M” จะทำให้ทราบถึงประสิทธิภาพของเทคนิคการสร้างสูตรสำเร็จการแก้สมการเมทริกซ์ในรูปรหัสคำสั่งเครื่อง

6.3.8 **โปรแกรม** “เล็ก 6MInv”



รูปที่ 6.8 แผนภูมิการทำงานอย่างง่ายของโปรแกรม “เล็ก 6MInv”

โปรแกรม “เล็ก 6MInv” เป็นโปรแกรมที่พัฒนาต่อจากโปรแกรม “เล็ก 6M” โดยเพิ่มเทคนิคการกลับเศษส่วนค่าสมาชิกในแนวทแยงมุขสำคัญ เพื่อทดสอบผลของเทคนิคนี้เมื่อใช้รวมกับเทคนิคการสร้างสูตรสำเร็จการแก้สมการเมทริกซ์ในรูปรหัสคำสั่งเครื่อง โดยรูปที่ 6.8 แสดงแผนภูมิการทำงานอย่างง่ายของโปรแกรม “เล็ก 6MInv”

จากรายละเอียดของโปรแกรมที่ทดสอบทั้งหมดสามารถสรุปเทคนิคที่ใช้เพื่อเพิ่มความเร็วในการจำลองวงจรได้ดังตารางที่ 6.1 ส่วนตารางที่ 6.2 แสดงถึงหน่วยความจำที่แต่ละโปรแกรมใช้เพื่อจำลองวงจร

เทคนิคที่ใช้	LEK6.0	LEK6S	LEK6R	LEK6RInv	LEK6I	LEK6IInv	LEK6M	LEK6MInv
การข้ามการคำนวณกับค่าศูนย์		✓	✓	✓	✓	✓	✓	✓
โครงสร้างข้อมูลแบบเมทริกซ์มากเลขศูนย์		✓	✓	✓	✓	✓	✓	✓
การเรียงลำดับใหม่			✓	✓	✓	✓	✓	✓
การกลับเศษส่วนค่าสมาชิกในแนวทแยงมุขสำคัญ				✓		✓		✓
การสร้างรหัสแบบแปลย่อย					✓	✓		
การสร้างรหัสคำสั่งเครื่อง							✓	✓

ตารางที่ 6.1 เทคนิคของเมทริกซ์มากเลขศูนย์ที่แต่ละโปรแกรมใช้เพื่อจำลองวงจร

หน่วยความจำที่ใช้	LEK6.0	LEK6S	LEK6R	LEK6RInv	LEK6I	LEK6IInv	LEK6M	LEK6MInv
เมทริกซ์ธรรมดา	✓							
เมทริกซ์มากเลขศูนย์		✓	✓	✓	✓	✓	✓	✓
รหัสแบบแปลย่อย					✓	✓		
รหัสคำสั่งเครื่อง							✓	✓

ตารางที่ 6.2 หน่วยความจำที่แต่ละโปรแกรมใช้เพื่อจำลองวงจร

6.4 ผลการทดสอบและวิจารณ์ผล

6.4.1 **ผลการทดสอบด้านเวลา**

การทดสอบด้านเวลาทำโดยให้โปรแกรม"เล็ก"ทั้ง 8 โปรแกรมจำลองการทำงานของวงจรทดสอบทั้ง 11 วงจร และจับเวลาการทำงาน โดยเริ่มนับเวลาตั้งแต่เข้าสู่วนรอบการแก้สมการวงจรในรอบการทำงานแรกจนกระทั่งสิ้นสุดการแก้สมการวงจรในรอบสุดท้าย ขั้นตอนการเก็บผลลัพธ์คำตอบ, การแสดงผลกราฟคำตอบ และการทำงานอื่นๆที่ไม่จำเป็นต่อหาผลคำตอบ จะไม่ถูกรวมในช่วงที่จับเวลานี้

ส่วนเวลาจะมีหน่วยเป็น Tick ซึ่งเทียบเท่ากับ $\frac{1}{1165}$ วินาที หรือประมาณ 0.858 ms โดยการวัดเวลาที่มีหน่วยละเอียดระดับนี้จำเป็นต้องใช้เทคนิคพิเศษ คือ การตั้งค่าตัวหารเวลาใน IC 8253-5 (Programmable Interval Timer)[16] ในเครื่องคอมพิวเตอร์ที่จะทดสอบ จากเดิมที่ปรกติค่าตัวหารเวลาเป็น 256 ซึ่งจะทำให้ 1 Tick เทียบเท่ากับ $\frac{1}{18.2}$ วินาที เปลี่ยนให้ค่าตัวหารเวลาเป็น 4 มีผลให้ Real Time Clock ของเครื่องวิ่งเร็วขึ้น 64 เท่าแต่ความเร็วในการคำนวณยังคงเดิม การตั้งค่าตัวหารเวลาใน IC 8253-5 สามารถทำได้โดยใช้คำสั่งภาษา Assembly ดังนี้

```
CLI              ; Clear Interrupt Flag
MOV  DX, 43h     ; เลือกส่งข้อมูลไปที่ Port เบอร์ 43h ( 8253-5 Control Port )
MOV  AL, 36h     ; เลือกให้ส่งข้อมูลไปที่ Counter 0 ของ IC 8253-5 เลือก Mode 3 Square
                   Wave Rate Generator
OUT  DX, AL      ; ส่งรหัส 36h ไปที่ Port เบอร์ 43h
NOP              ; รอให้ส่งข้อมูลเรียบร้อย
MOV  DX, 40h     ; เลือกส่งข้อมูลไปที่ Port เบอร์ 40h ( 8253-5 Counter 0 )
XOR  AX, AX      ; กำหนดค่า AX เป็น 0
OUT  DX, AL      ; ส่งรหัส 00h ไปที่ Port เบอร์ 40h
NOP              ; รอให้ส่งข้อมูลเรียบร้อย
MOV  AL, 04h     ; เลือกเลือกค่าตัวหารเวลาเป็น 4
OUT  DX, AL      ; ส่งรหัส 04h ไปที่ Port เบอร์ 40h
STI              ; Set Interrupt Flag
```

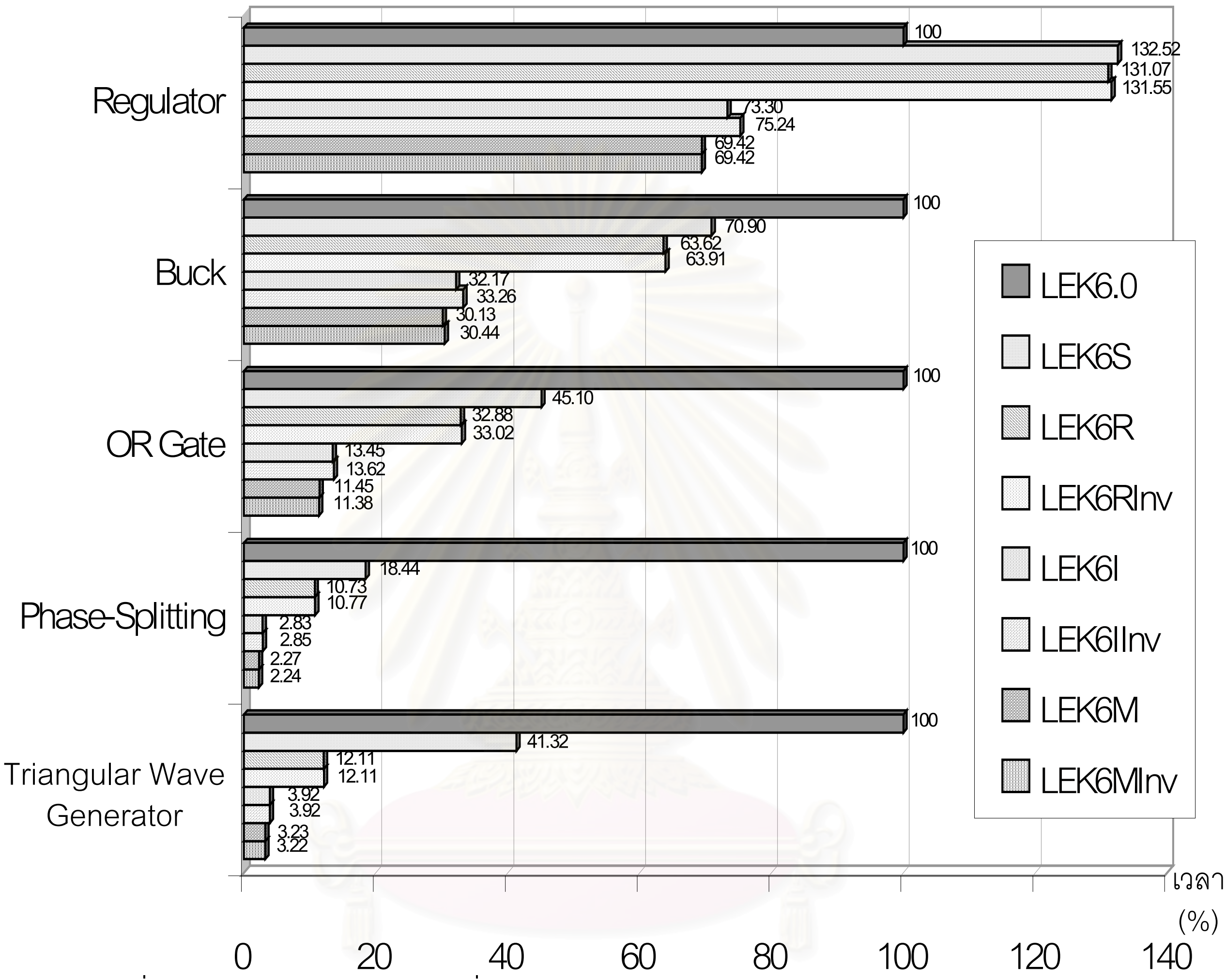
ผลการทดสอบด้านเวลาทั้งหมดแสดงในตารางที่ 6.3

วงจรทดสอบ	จำนวนตัวแปร	ดัชนีมากเลขศูนย์*	เวลาที่ใช้ในการจำลองวงจร (Ticks)							
			LEK6.0	LEK6S	LEK6R	LEK6RInv	LEK6I	LEK6IInv	LEK6M	LEK6MInv
Regulator	4	62.50%	206	273	270	271	151	155	143	143
Buck	7	32.70%	27347	19388	17399	17477	8797	9096	8239	8325
OR Gate	18	22.50%	9662	4358	3177	3190	1300	1316	1106	1100
Phase-Splitting	33	11.50%	41251	7606	4426	4443	1167	1177	937	926
Triangular Wave Generator	48	11.63%	493684	203967	59775	59808	19376	19347	15929	15893
RC3	3	77.80%	264	403	404	406	230	237	221	225
RC5	5	52.00%	609	779	677	680	346	354	329	327
RC10	10	28.00%	3047	1681	1326	1336	639	652	595	594
RC30	30	9.78%	62205	6776	5157	5181	1990	2033	1853	1846
RC50	50	5.92%	281680	13726	10772	10812	3284	3350	3032	3012
RC100	100	2.98%	2155238	39549	32474	32540	6778	6899	6264	6260

$$\underset{(\text{Sparsity Index})}{{}^{*}\text{ดัชนีมากเลขศูนย์}} = \frac{\text{จำนวนสมาชิกที่มีค่าไม่เป็นศูนย์}}{\text{จำนวนสมาชิกทั้งหมดในเมทริกซ์}} \times 100\%$$

ตารางที่ 6.3 ผลการทดสอบด้านเวลาในการจำลองวงจรไฟฟ้า

สำหรับการทดสอบวงจรที่มีใช้จริงในทางปฏิบัติ 5 วงจรแรก ถ้ากำหนดให้ระยะเวลาที่โปรแกรม “เล็ก 6.0” ใช้ในการจำลองแต่ละวงจรถือเป็น 100% จะสามารถเขียนกราฟแสดงสัดส่วนเวลาในการจำลองวงจรของโปรแกรมต่างๆเทียบกับโปรแกรม “เล็ก 6.0” ได้ดังรูปที่ 6.9



รูปที่ 6.9 กราฟแสดงสัดส่วนเวลาที่โปรแกรมต่างๆใช้ในการจำลองวงจรทดสอบเทียบกับโปรแกรม “เล็ก 6.0”

จากรูปที่ 6.9 จะให้ได้ชัดเจนว่าโปรแกรม “เล็ก 6.0” ใช้เวลาในการจำลองวงจรนานที่สุด ตามด้วยโปรแกรม “เล็ก 6S”, “เล็ก 6RInv”, “เล็ก 6R”, “เล็ก 6IInv”, “เล็ก 6I”, “เล็ก 6M” และ “เล็ก 6MInv” ตามลำดับ ยกเว้นในกรณีของวงจร Regulator ซึ่งโปรแกรม “เล็ก 6.0” ใช้เวลาจำลองวงจรน้อยกว่าโปรแกรม “เล็ก 6S”, “เล็ก 6R” และ “เล็ก 6RInv”

เมื่อเปรียบเทียบเวลาที่ใช้คำนวณของโปรแกรม “เล็ก 6.0” กับ “เล็ก 6S” จะเห็นแนวโน้มว่าเทคนิคสำหรับเมทริกซ์มากเลขศูนย์ช่วยลดเวลาการคำนวณลงได้โดยขึ้นกับดัชนีมากเลขศูนย์ ยิ่งค่าดัชนีมากเลขศูนย์น้อยจะทำให้ยิ่งใช้เวลาคำนวณลดลงไปด้วยซึ่งสอดคล้องกับทฤษฎีของเทคนิคสำหรับเมทริกซ์มากเลขศูนย์ และถ้าค่าดัชนีมากเลขศูนย์มีค่ามากๆอาจทำให้โปรแกรมที่ใช้เทคนิคนี้ใช้

เวลาคำนวณมากกว่าโปรแกรมธรรมดาที่ไม่ได้เทคนิคสำหรับเมทริกซ์มากเลขศูนย์ด้วย ดังเช่นในกรณีของ วงจร Regulator ทั้งนี้เพราะการจัดการกับโครงสร้างการเก็บข้อมูลแบบเมทริกซ์มากเลขศูนย์ย่อมต้องมีกระบวนการมากกว่าการจัดการกับโครงสร้างข้อมูลธรรมดาอย่างแน่นอน ส่วนในกรณีของวงจร Phase-Splitting และวงจร Triangular Wave Generator ที่มีค่าดัชนีมากเลขศูนย์ใกล้เคียงกัน แต่โปรแกรม “เล็ก 6S” ใช้เวลาจำลองวงจร Triangular Wave Generator คิดเป็นสัดส่วนเปรียบเทียบมากกว่าวงจร Phase-Splitting ถึงประมาณหนึ่งเท่าตัวนั้น เป็นเพราะว่าขั้นตอนการแยกตัวประกอบแอล-ยูทำให้เกิด Fill-in จำนวนมากในเมทริกซ์สัมประสิทธิ์ของสมการวงจร Triangular Wave Generator มีผลให้ค่าดัชนีมากเลขศูนย์เพิ่มขึ้นจนเป็น 52.08% ทำให้ประสิทธิภาพของเทคนิคสำหรับเมทริกซ์มากเลขศูนย์ลดลงไป

เมื่อเปรียบเทียบเวลาที่ใช้ของโปรแกรม “เล็ก 6S” กับ “เล็ก 6R” จะพบว่าเทคนิคการเรียงลำดับใหม่ตามวิธีของ Markowitz ช่วยลดเวลาการคำนวณลงได้โดยขึ้นกับวงจรทดสอบเช่นกัน สาเหตุที่โปรแกรม “เล็ก 6R” ใช้เวลาคำนวณน้อยกว่าเป็นเพราะเทคนิคการเรียงลำดับใหม่ช่วยทำให้เกิด Fill-in น้อยลง ทำให้ค่าดัชนีมากเลขศูนย์เพิ่มขึ้นไม่มากนักเมื่อแยกตัวประกอบแอล-ยู ส่งผลให้เทคนิคสำหรับเมทริกซ์มากเลขศูนย์มีประสิทธิภาพมากยิ่งขึ้นด้วย

ในการทดสอบประสิทธิ์ภาพของเทคนิคการกลับเศษส่วนค่าสมาชิกในแนวทแยงมุขสำคัญ จะต้องเปรียบเทียบผลการทดสอบของโปรแกรม 3 คู่ คือ “เล็ก 6R” กับ “เล็ก 6RInv”, “เล็ก 6I” กับ “เล็ก 6IInv” และ “เล็ก 6M” กับ “เล็ก 6MInv” จะพบว่าสำหรับเทคนิคนี้เมื่อใช้ร่วมกับเทคนิคอื่นจะทำให้โปรแกรมใช้เวลาการคำนวณมายิ่งขึ้นไปอีก ยกเว้นเมื่อใช้ร่วมกับเทคนิคการสร้างสูตรสำเร็จการแก้สมการเมทริกซ์ด้วยรหัสคำสั่งเครื่องเท่านั้น ทั้งนี้เป็นเพราะว่าการเปลี่ยนการหารเป็นการคูณนั้นช่วยลดเวลาการคำนวณลงเพียงเล็กน้อย ทำให้เวลาที่ลดลงน้อยกว่าเวลาที่เสียไปในขั้นตอนต่างๆที่ทำเพื่อนำเทคนิคนี้มาใช้ แต่การแก้สมการเมทริกซ์ด้วยรหัสคำสั่งเครื่องจะเสียเวลาทำขั้นตอนที่เพิ่มขึ้นน้อยมากจึงทำให้โปรแกรม “เล็ก 6MInv” คำนวณได้เร็วกว่าโปรแกรม “เล็ก 6M” อยู่เล็กน้อย

สำหรับโปรแกรม “เล็ก 6I” จะใช้เวลาจำลองวงจรน้อยกว่าโปรแกรม “เล็ก 6R” ประมาณ 2 ถึง 4 เท่า เพราะเทคนิคการสร้างสูตรสำเร็จการแก้สมการเมทริกซ์ทำให้ลดเวลาการคำนวณต่างๆดังนี้

1. สามารถข้ามขั้นตอนการเรียงลำดับใหม่ได้ เนื่องจากสูตรสำเร็จการแก้สมการเมทริกซ์ไม่ได้อ้างอิงตำแหน่งของสมาชิกด้วยแถวและคอลัมน์ในเมทริกซ์ แต่จะอ้างอิงด้วยตำแหน่งในแถวลำดับที่เก็บข้อมูลเป็นสำคัญ ดังนั้นการคำนวณด้วยสูตรสำเร็จการแก้สมการเมทริกซ์จึงเสมือนได้ทำขั้นตอนการเรียงลำดับใหม่โดยอัตโนมัติ
2. ไม่จำเป็นต้องเปรียบเทียบตำแหน่งของสมาชิกในเมทริกซ์เพื่อการคำนวณในขั้นตอนการแยกตัวประกอบแอล-ยู
3. ไม่จำเป็นต้องมีการวนรอบการทำงานสำหรับขั้นตอนคำนวณต่างๆ

4. สามารถใช้ Register ภายใน FPU สำหรับเป็นตัวแปรชั่วคราวในการคำนวณได้ ทำให้ลดจำนวนครั้งการอ่านและเขียนข้อมูลกับหน่วยความจำภายนอกหน่วยประมวลผลกลาง

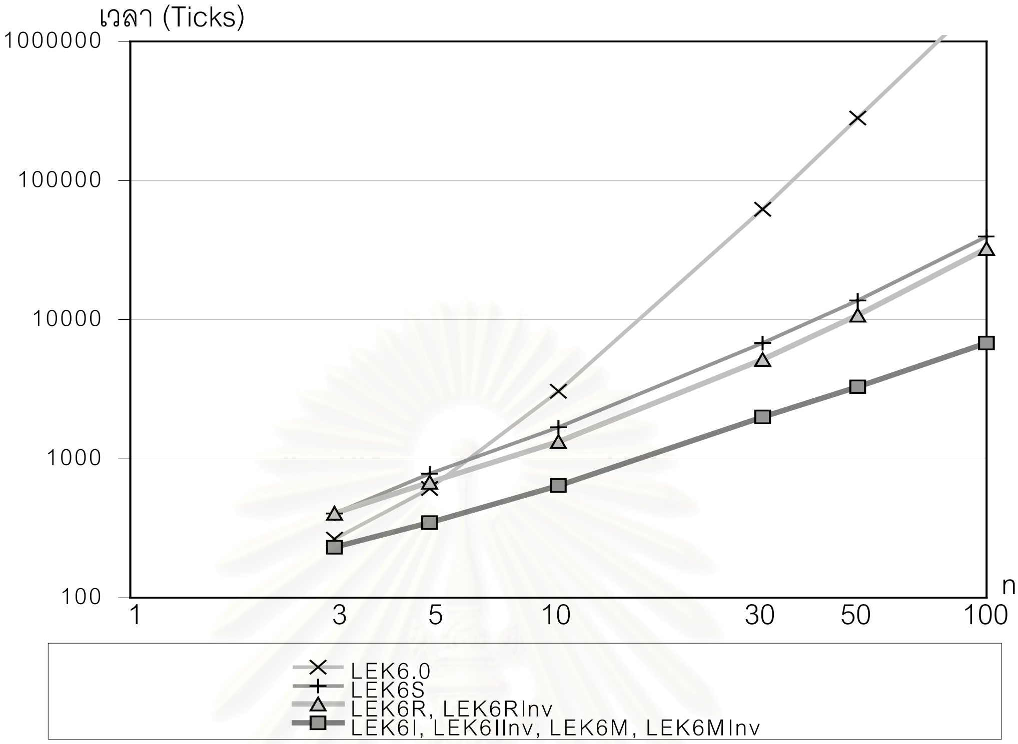
และหากใช้เทคนิคการสร้างสูตรสำเร็จการแก้สมการเมทริกซ์ด้วยรหัสคำสั่งเครื่องแล้วจะสามารถลดเวลาคำนวณลงได้อีก เนื่องจากหน่วยประมวลผลกลางสามารถทำงานตามรหัสคำสั่งเครื่องได้ทันทีโดยไม่ต้องเสียเวลาวนรอบเพื่อตีความรหัสคำสั่งเลย แต่จากผลการทดสอบปรากฏว่าโปรแกรม “เล็ก 6M” ใช้เวลาคำนวณน้อยกว่าโปรแกรม “เล็ก 6I” อยู่เพียงประมาณ 5-20% เท่านั้น ทั้งที่ควรจะใช้เวลาน้อยลงถึงประมาณ 10 - 12 เท่าเมื่อเปรียบเทียบตามรหัสคำสั่งเครื่องที่หน่วยประมวลผลกลางต้องทำงานจริง ดังนั้น จึงได้ทดสอบโปรแกรม “เล็ก 6I” และ “เล็ก 6M” เพิ่มเติม โดยแยกจับเวลาเฉพาะการ Load Sparse Matrix, และการแก้สมการด้วยรหัสแบบแปลย่อยและรหัสคำสั่งเครื่อง ซึ่งได้ผลตามตารางที่ 6.4

วงจรที่ทดสอบ	เล็ก 6I			เล็ก 6M			เล็ก 6I (Solve) / เล็ก 6M (Solve)
	Load Sparse Matrix	Solve by Interpretive Code	Load Matrix / Solve	Load Sparse Matrix	Solve by Machine Code	Load Matrix / Solve	
Regulator	88	14	6.29	87	7	12.43	2.00
Buck	5243	960	5.46	5172	387	13.36	2.48
OR Gate	936	247	3.79	919	50	18.38	4.94
Phase-Splitting	776	312	2.49	778	71	10.96	4.39
Triangular Wave Generator	13487	4113	3.28	13701	718	19.08	5.73
RC3	141	20	7.05	141	9	15.67	2.22
RC5	248	36	6.89	238	16	14.88	2.25
RC10	472	54	8.74	470	27	17.41	2.00
RC30	1608	249	6.46	1604	86	18.65	2.90
RC50	2541	435	5.84	2568	140	18.34	3.11
RC100	5192	897	5.79	5176	283	18.29	3.17

ตารางที่ 6.4 ผลการทดสอบเวลาที่ใช้คำนวณในส่วน Load Matrix และส่วนแก้สมการของโปรแกรม “เล็ก 6I” และ “เล็ก 6M”

จะเห็นว่าเวลาการคำนวณส่วนใหญ่กลับเป็นการทำงานในขั้นตอน Load Sparse Matrix ซึ่งใช้เวลาคงที่ในทุกโปรแกรม ทั้งยังใช้เวลามากกว่าการแก้สมการด้วยรหัสแบบแปลย่อยและรหัสคำสั่งเครื่องหลายเท่าตัว จึงทำให้ดูคล้ายว่าเทคนิคการแก้สมการด้วยรหัสคำสั่งเครื่องใช้เวลาน้อยกว่าการใช้รหัสแบบแปลย่อยเพียงเล็กน้อย ทั้งที่ความจริงใช้เวลาคำนวณน้อยกว่าประมาณ 2 - 6 เท่า

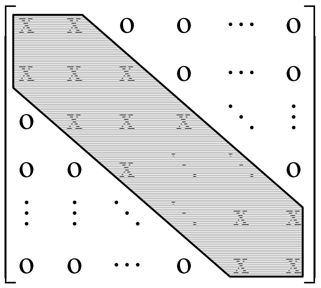
สำหรับผลการทดสอบด้านเวลาของวงจร RC Ladder สามารถนำมาวาดกราฟได้ดังรูปที่ 6.10



รูปที่ 6.10 กราฟแสดงเวลาที่โปรแกรม “เล็ก” แต่ละโปรแกรมใช้ในการจำลองวงจร RC Ladder

จากกราฟรูปที่ 6.10 ซึ่งแกนเป็นสัดส่วนลอการิทึมและแกนนอนเป็นค่าจำนวนตัวแปรอิสระ (n)ในวงจร RC Ladder สังเกตได้ว่าโปรแกรม “เล็ก 6.0” ใช้เวลาจำลองวงจรมากที่สุดโดยมีลำดับ (Order) เป็น n^3 ซึ่งสอดคล้องกับหลักการแก้สมการเมทริกซ์โดยวิธีแยกตัวประกอบแอล-ยู ส่วนเวลาที่โปรแกรม “เล็ก 6S”, “เล็ก 6R” และ “เล็ก 6RInv” ใช้จำลองวงจรมีลำดับอยู่ระหว่าง n กับ n^2 ส่วนโปรแกรม “เล็ก 6I”, “เล็ก 6IINv”, “เล็ก 6M” และ “เล็ก 6MInv” ใช้เวลาจำลองวงจรคิดลำดับเป็น n

โดยตามทฤษฎีแล้วโปรแกรมทั้งหมดควรใช้เวลาจำลองวงจรคิดลำดับเป็น n^3 แต่เนื่องจากวงจร RC Ladder มีลักษณะพิเศษคือ สมาชิกที่มีค่าไม่เป็นศูนย์ในเมทริกซ์สัมประสิทธิ์จะปรากฏอยู่เฉพาะตามแนวทแยงมุมสำคัญเท่านั้น ซึ่งเรียกว่า Band Triangular Form[17] ดังแสดงในรูปที่ 6.11 เทคนิคสำหรับเมทริกซ์มากเลขศูนย์และเทคนิคการสูตรสำเร็จการแก้สมการเมทริกซ์จึงสามารถลดขั้นตอนการคำนวณที่ไม่จำเป็นทิ้งไปและลดลำดับการคำนวณลงมาจนเข้าใกล้ n ได้



รูปที่ 6.11 ตำแหน่งของสมาชิกที่ไม่เป็นศูนย์ในเมทริกซ์สัมประสิทธิ์ของสมการวงจร RC Ladder

6.4.2 ผลการทดสอบด้านการใช้หน่วยความจำ

ขนาดหน่วยความจำที่ถูกนำมาเปรียบเทียบในการทดสอบนี้มี 2 คู่คือ

1. หน่วยความจำที่ใช้เก็บเมทริกซ์ธรรมดา กับ หน่วยความจำที่ใช้เก็บเมทริกซ์มากเลขศูนย์
2. หน่วยความจำที่ใช้เก็บรหัสแบบแปลย่อย กับ หน่วยความจำที่ใช้เก็บรหัสคำสั่งเครื่อง

ขนาดหน่วยความจำที่เก็บเมทริกซ์ธรรมดาคำนวณได้โดยเท่ากับ 8 * n^2 ไบต์ โดยที่ n คือจำนวนตัวแปรอิสระในวงจร และขนาดหน่วยความจำที่เก็บเมทริกซ์มากเลขศูนย์เท่ากับ (16*nzero)+(6*n) ไบต์ โดยที่ nzero คือจำนวนสมาชิกที่ไม่เป็นศูนย์ในเมทริกซ์ (หัวข้อ 4.2 ในบทที่ 4)

ส่วนขนาดหน่วยความจำที่ใช้เก็บรหัสแบบแปลย่อยและรหัสคำสั่งเครื่อง ซึ่งมีทั้งแบบธรรมดาและแบบที่ใช้เทคนิคการกลับเศษส่วนของสมาชิกในแนวทแยงมุขสำคัญ จะหาได้โดยให้โปรแกรม “เล็ก 6I”,”เล็ก 6IInv”,”เล็ก 6M” และ “เล็ก 6MInv” จำลองการทำงานของแต่ละวงจรแล้วจึงวัดขนาดหน่วยความจำที่ใช้เก็บรหัสทั้งสี่แบบ

ผลการทดสอบโปรแกรมด้านการใช้หน่วยความจำแสดงในตารางที่ 6.5 ซึ่งจะพบว่า ขนาดของหน่วยความจำที่เก็บโครงสร้างข้อมูลเมทริกซ์มากเลขศูนย์จะน้อยกว่าโครงสร้างข้อมูลเมทริกซ์ธรรมดา โดยขึ้นกับค่าดัชนีมากเลขศูนย์เป็นสำคัญ ถ้าค่าดัชนีมากเลขศูนย์น้อยกว่า $\left(50-\frac{37.5}{n}\right)\%$ จะสามารถลดขนาดที่ใช้เก็บข้อมูลลงได้ ในทางกลับกันถ้าค่าดัชนีมากเลขศูนย์มีค่ามากกว่า $\left(50-\frac{37.5}{n}\right)\%$ จะทำให้เมทริกซ์มากเลขศูนย์ต้องใช้พื้นที่หน่วยความจำมากกว่าเมทริกซ์ธรรมดาด้วย แต่ปัญหาสำคัญเรื่องหน่วยความจำไม่เพียงพอจะเกิดขึ้นกับการจำลองวงจรขนาดใหญ่ซึ่งมักจะมีค่าดัชนีมากเลขศูนย์น้อย ดังนั้นโครงสร้างข้อมูลแบบเมทริกซ์มากเลขศูนย์จึงจะช่วยให้โปรแกรมสามารถจำลองวงจรขนาดใหญ่กว่าโปรแกรมที่ใช้โครงสร้างเมทริกซ์แบบธรรมดา

ส่วนหน่วยความจำที่ใช้เก็บรหัสคำสั่งเครื่องจะมีขนาดใหญ่กว่าหน่วยความจำที่ใช้เก็บรหัสแบบแปลย่อยอยู่ประมาณ 1.9 - 2.6 เท่าแต่สามารถเพิ่มความเร็วในการแก้สมการเมทริกซ์ได้ถึง 2 - 6 เท่าซึ่งถือได้ว่าคุ้มค่า แต่จะควรต้องลดเวลาในขั้นตอน Load Sparse Matrix ลงอีกมากเพื่อให้เห็นผลของเทคนิคการแก้สมการเมทริกซ์ด้วยรหัสคำสั่งเครื่องได้อย่างชัดเจน

สำหรับกรณีที่เพิ่มเทคนิคการกลับเศษส่วนของสมาชิกในแนวทแยงมุขสำคัญจะทำให้รหัสแบบแปลย่อยและรหัสคำสั่งเครื่องมีขนาดใหญ่ขึ้น หรือหมายความว่ามีการคำนวณมากขึ้น ซึ่งสอดคล้องการผลการทดสอบด้านเวลาที่โปรแกรม “เล็ก 6IInv” คำนวณช้ากว่าโปรแกรม “เล็ก 6I”

วงจรทดสอบ	จำนวนตัวแปร (n)	ดัชนีมากเลขศูนย์	หน่วยความจำ (ไบต์)		เมทริกซ์มากเลขศูนย์ / เมทริกซ์ธรรมดา	หน่วยความจำ (ไบต์)				รหัสคำสั่งเครื่อง / รหัสแบบแปลย่อย
			เมทริกซ์ธรรมดา	เมทริกซ์มากเลขศูนย์		รหัสแบบแปลย่อย	รหัสแบบแปลย่อย (กลับเศษส่วน)	รหัสคำสั่งเครื่อง	รหัสคำสั่งเครื่อง (กลับเศษส่วน)	
Regulator	4	62.50%	128	184	143.75%	96	120	248	312	2.58
Buck	7	32.70%	392	298	76.02%	144	192	358	488	2.49
OR Gate	18	22.50%	2592	1276	49.23%	1518	1248	3114	3398	2.05
Phase-Splitting	33	11.50%	8712	2198	25.23%	2562	2142	5206	5796	2.03
Triangule Wave Generator	48	11.63%	18432	4576	24.83%	9252	9540	17290	18106	1.89
RC3	3	77.80%	72	130	180.56%	66	84	172	222	2.61
RC5	5	52.00%	200	238	119.00%	126	156	324	402	2.57
RC10	10	28.00%	800	508	63.50%	276	336	704	852	2.55
RC30	30	9.78%	7200	1588	22.06%	876	1056	2224	2652	2.54
RC50	50	5.92%	20000	2668	13.34%	1476	1776	3744	4452	2.54
RC100	100	2.98%	80000	5368	6.71%	2976	3576	7544	8952	2.53

ตารางที่ 6.5 ผลการทดสอบด้านการใช้หน่วยความจำในการจำลองวงจรไฟฟ้า

บทที่ 7

สรุปผลและข้อเสนอแนะ

7.1 <u>สรุปผลการทำวิทยานิพนธ์</u>

เทคนิคของเมทริกซ์มากเลขศูนย์ เป็นเทคนิคที่ช่วยเพิ่มประสิทธิภาพในการแก้สมการเมทริกซ์ที่มีเมทริกซ์สัมประสิทธิ์เป็นเมทริกซ์มากเลขศูนย์ ซึ่งเทคนิคนี้เป็นประโยนช์อย่างยิ่งสำหรับงานหลากหลายประเภท แต่สำหรับวิทยานิพนธ์นี้ ได้นำเทคนิคของเมทริกซ์มากเลขศูนย์มาประยุกต์ใช้กับโปรแกรมจำลองการทำงานของวงจรไฟฟ้า

เทคนิคของเมทริกซ์มากเลขศูนย์ที่นำมาใช้ในการปรับปรุงโปรแกรม”เล็ก 6.0” นี้ได้แก่

1. **การข้ามการคำนวณกับค่าศูนย์** เป็นเทคนิคที่ใช้ประโยชน์จากหลักคณิตศาสตร์ง่ายๆที่ทราบกันว่าการนำค่าศูนย์ไปบวกหรือลบกับค่าตัวเลขใดก็จะได้ค่าตัวเลขเดิมนั้น และค่าศูนย์เมื่อคูณหรือหารด้วยค่าใดๆก็จะได้ค่าศูนย์ทุกครั้ง ทำให้สามารถลดการคำนวณในการแก้สมการเมทริกซ์ลงไปได้มาก หากเมทริกซ์สัมประสิทธิ์ของสมการนั้นเป็นเมทริกซ์มากเลขศูนย์
2. **โครงสร้างการเก็บข้อมูล** เป็นเทคนิคการจัดเก็บข้อมูลของเมทริกซ์ที่มีสมาชิกส่วนใหญ่มีค่าเป็นศูนย์ โดยนอกจากต้องการให้ใช้เนื้อที่หน่วยความจำในการเก็บข้อมูลให้น้อยลงแล้ว ยังต้องการให้มีคุณสมบัติอีกหลายประการที่จะช่วยให้การแก้สมการเมทริกซ์ด้วยวิธีแยกตัวประกอบแอล-ยูทำได้สะดวกรวดเร็วขึ้น
3. **การเรียงลำดับใหม่** เป็นเทคนิคการสลับแถวและคอลัมน์ของเมทริกซ์สัมประสิทธิ์ก่อนขั้นตอนการแยกตัวประกอบแอล-ยู ซึ่งสามารถลดจำนวน Fill-ins ที่เกิดขึ้นใหม่ลงได้ และเมื่อใช้รวมกับเทคนิคการข้ามการคำนวณกับค่าศูนย์จะทำให้ลดจำนวนครั้งการคำนวณลงไปได้อีกมาก จึงถือได้ว่าเป็นเทคนิคที่ช่วยเสริมประสิทธิภาพของเทคนิคการข้ามการคำนวณกับค่าศูนย์
4. **การกลับเศษส่วนค่าสมาชิกในแนวทแยงมุขสำคัญ** เป็นเทคนิคเสริมอย่างหนึ่งที่พยายามลดจำนวนครั้งการหารลง โดยแปรเปลี่ยนให้กลายเป็นการคูณแทน เพราะหน่วยประมวลผลกลางสามารถคำนวณคำสั่งคูณได้เร็วกว่าคำสั่งหารถึง 10 เท่า แต่เทคนิคใช้ได้ผลกับเทคนิคการสร้างสูตรสำเร็จการแก้สมการด้วยรหัสคำสั่งเครื่องเท่านั้น

5. **การสร้างรหัสแบบแปลคำสั่ง** เป็นเทคนิคที่มีประโยชน์ในกรณีที่ต้องการสมการเมทริกซ์เดิมซ้ำกันหลายๆครั้ง โดยที่สมาชิกที่มีค่าไม่เป็นศูนย์ในเมทริกซ์สัมประสิทธิ์ยังอยู่ในตำแหน่งเดิมทุกๆครั้ง จึงสามารถใช้รหัสแบบแปลคำสั่งที่สร้างขึ้นทำการแก้สมการเมทริกซ์นั้นให้ได้ผลลัพธ์อย่างถูกต้องและรวดเร็วกว่า
6. **การสร้างรหัสคำสั่งเครื่อง** เป็นเทคนิคที่พัฒนาต่อมาจากเทคนิคการสร้างรหัสแบบแปลคำสั่ง โดยเปลี่ยนจากรหัสที่จะต้องแปลคำสั่งก่อนการคำนวณให้กลายเป็นรหัสคำสั่งที่เครื่องคอมพิวเตอร์สามารถนำไปประมวลผลได้ทันที จึงสามารถลดเวลาในการคำนวณลงได้อีกมาก

เทคนิคที่ 1 ถึง 4 ที่ได้กล่าวมาได้นำมาใช้ปรับปรุงโปรแกรม ”เล็ก 6.0” จนกลายเป็นโปรแกรม “เล็ก 6.0RInv” และได้พัฒนาต่อเป็นโปรแกรม ”เล็ก 6.0IInv” ด้วยการเพิ่มเทคนิคการสร้างรหัสแบบแปลคำสั่ง นอกจากนี้ยังได้พัฒนาโปรแกรม ”เล็ก 6.0MInv” ที่ใช้เทคนิคการสร้างรหัสคำสั่งเครื่องแทน ซึ่งถือว่าเป็นโปรแกรม “เล็ก” รุ่นที่จำลองวงจรได้เร็วที่สุดในจำนวนโปรแกรมที่พัฒนาขึ้นมาทั้งหมดด้วย

7.2 ข้อเสนอแนะ

หลังจากที่ได้ทดลองนำเอาเทคนิคต่างๆที่กล่าวมาแล้วไปปรับปรุงโปรแกรม ”เล็ก 6.0” จนได้โปรแกรม”เล็ก” รุ่นใหม่ที่สามารถลดเวลาในการคำนวณลงได้หลายเท่าตัว ซึ่งทำให้โปรแกรมนี้มีประโยชน์มากในการใช้งานจริงโดยเฉพาะงานทางด้านอิเล็กทรอนิกส์กำลังซึ่งส่วนใหญ่จะต้องจำลองการทำงานของวงจรเป็นเวลานาน เช่น วงจรทางด้านการสวิทช์ (Switching) เป็นต้น หรือในงานด้านวงจรผสมอนาล็อกดิจิตอล (Analog-Digital Mix Signal Circuit) ซึ่งจะมีรูปแบบการทดสอบวงจรจำนวนมาก แต่ยังมีเทคนิคอื่นอีกที่สามารถปรับปรุงโปรแกรม”เล็ก”ให้มีประสิทธิภาพดียิ่งขึ้นได้ ดังตัวอย่างเช่น

1. นำเทคนิคขั้นตอนวิธีเมทริกซ์แคช [7] มาใช้ร่วมกับเทคนิคมากเลขศูนย์เพื่อเพิ่มความเร็วในการคำนวณผลอีก
2. ควรตรวจสอบด้วยว่าสมาชิกที่ไม่เป็นศูนย์ในเมทริกซ์สัมประสิทธิ์ตัวใดที่มีค่าเป็น 1 หรือ –1 เนื่องจากในเมทริกซ์สัมประสิทธิ์ของสมการวงจรจะมีค่า 1 และ –1 อยู่มากพอสมควร ซึ่งจะสามารถข้ามการคำนวณการคูณและการหารกับค่า 1 หรือ –1 ได้อีก
3. ปรับปรุงขั้นตอน Load Sparse Matrix ในส่วนการคำนวณค่าที่จะใส่ลงในเมทริกซ์ ให้ใช้เวลาน้อยลงกว่าเดิม

รายการอ้างอิง

[1] Gielen, G.; Walscharts, H.; and Sansen, W. ISAAC : A Symbolic Simulator for Analog Integrated Circuit. IEEE J. solid-state Circuits 24. 6 (Dec. 1989): 1587-1597.

[2] สุเจตน์ จันทรังษ์. โปรแกรมวิเคราะห์เน็ทเวิร์คแบบสัญลักษณ์. การประชุมวิชาการทางวิศวกรรมไฟฟ้า ครั้งที่ 14 14 (2534): 5-29 ถึง 5-32.

[3] สุเจตน์ จันทรังษ์ และ สิทธิชัย โภไคยอุดม. โปรแกรมวิเคราะห์วงจรแบบสัญลักษณ์. การประชุมวิชาการทางวิศวกรรมไฟฟ้า ครั้งที่ 16 16 (2536): 521-526.

[4] Nagel, L.W. SPICE2: A computer program to simulate semiconductor circuits. ERL Memo ERL-M520 University of California : Berkeley, May 1975.

[5] เอกชัย ลีลารัศมี. การใช้ไมโครคอมพิวเตอร์คำนวณหาผลตอบสนองเชิงเวลาของวงจรเชิงเส้นทั่วไป, การประชุมวิชาการทางวิศวกรรมไฟฟ้าครั้งที่ 7 16 (2527): เล่ม 3 : คอมพิวเตอร์ ค-131 ถึง ค-141.

[6] เอกชัย ลีลารัศมี และ สุริยัน ติษยาธิคม. รายงานฉบับสมบูรณ์ การพัฒนาโปรแกรมไมโครคอมพิวเตอร์สำหรับวิเคราะห์วงจรไฟฟ้าเชิงเส้น. ภาควิชาวิศวกรรมไฟฟ้า คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย, ตุลาคม 2529.

[7] เมธี หวังคุณธรรม. ขั้นตอนวิธีในการลดเวลาสำหรับการจำลองทางเวลาของวงจรเชิงเส้นแบบท่อนใน "เล็ก". วิทยานิพนธ์ปริญญามหาบัณฑิต ภาควิชาวิศวกรรมไฟฟ้า บัณฑิตวิทยาลัย จุฬาลงกรณ์มหาวิทยาลัย, 2539.

[8] Ho, C.; Ruehli, A.E.; and Brennan, P.A. The Modified Nodal Approach to Network Analysis. IEEE Transaction on Circuits and System CAS-25 (1975): 504-509.

[9] Chua, L.O.; and Lin, P.M. Computer Aided Analysis of Electronic Circuits : Algorithms and Computational Techniques. (n.p.):Prentice Hall, 1975.

[10] Eisenstat S. C. Yale Sparse Matrix Package. Research Reports 112, Connecticut: Dep. Computer Sciences, Yale Univ., 1975.

[11] Markowitz, H.M. The Elimination Form of the Inverse and its Application to Linear Programming. Management Science 3 (1957): 255-269.

[12] Berry, R.D. An Optimal Ordering of Electronic Circuit Equations for a Sparse Matrix Solution. IEEE Trans. Circ. Theory 18.1 (1971): 40-50.

[13] Jan Ogrodzki. Circuit Simulation Methods and Algorithms. Florida:CRC Press, 1994.

[14] Intel Corporation. Intel Architecture Software Developer's Manual. Vol. 1 : Basic Architecture. (n.p.), 1999.

[15] Giles, William B. Assembly Language Programmig for the Intel 80XXX Family. Singapore : Macmillan, 1991.

[16] IBM Corporation. IBM Personal Computer XT Technical Reference manual. (n.p.), 1983.

[17] Tewarson, R. P. Sparse Matrices. New York : Academic Press, 1973.

[18] ซีเอ็ดยูเคชั่น. คู่มือ/เทียบเบอร์ไอซี TTL. กรุงเทพมหานคร : ซีเอ็ดยูเคชั่น, 2526.

[19] Pinel J.F.; and Blostein, M. L. Computer techniques for the frequency analysis of linear networks. IEEE Proceedings (1967): 1810-1819.

ภาคผนวก

ภาคผนวก ก

วงจรที่ใช้ทดสอบประสิทธิ์ภาพของโปรแกรม

วงจรที่ใช้ทดสอบประสิทธิ์ภาพของโปรแกรม

เนื้อหาในบทที่ 6 เป็นการทดสอบโปรแกรม “เล็ก 6.0” และโปรแกรม “เล็ก” ที่ได้เพิ่มเทคนิคสำหรับเมทริกซ์มากเลขศูนย์ โดยตัวอย่างวงจรไฟฟ้าที่ใช้ทดสอบมีทั้งสิ้น 11 วงจร โดยแบ่งได้เป็น

1. วงจรที่มีใช้จริงในทางปฏิบัติ 5 วงจร ได้แก่ วงจร Regulator, วงจร Buck, วงจร OR Gate, วงจร Phase-Splitting และวงจร Triangular Wave Generator ทั้ง 5 วงจรถือเป็นตัวแทนวงจรที่ใช้ทดสอบเพื่อหาประสิทธิภาพของโปรแกรม “เล็ก” เมื่อนำไปใช้จำลองการทำงานของวงจรจริงในทางปฏิบัติ
2. วงจร RC Ladder 6 วงจรที่มีจำนวนปมในวงจรเป็น 3, 5, 10, 30, 50 และ 100 ปม เป็นวงจรไฟฟ้าที่มิได้นำมาใช้งานจริง แต่ถูกนำมาทดสอบเพื่อหาแนวโน้มประสิทธิภาพของโปรแกรมเมื่อเพิ่มจำนวนปมในวงจรมากขึ้นเรื่อยๆ จากวงจรขนาดเล็ก ไปจนถึงวงจรขนาดใหญ่

รายละเอียดของเมทริกซ์สัมประสิทธิ์ของสมการวงจรทดสอบทั้งหมด แสดงในตารางที่ ก.1

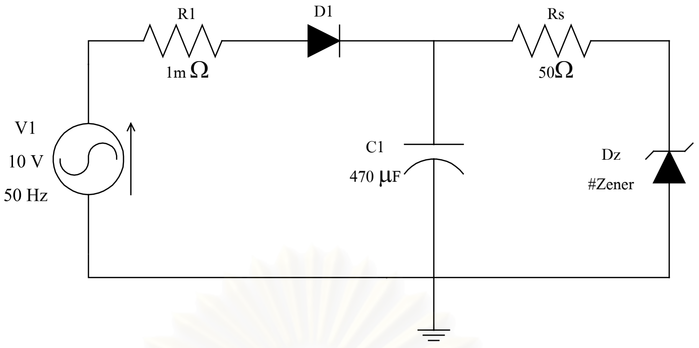
วงจรทดสอบ	จำนวนตัวแปรอิสระ	จำนวนสมาชิกที่ไม่เป็นศูนย์	ดัชนีมากเลขศูนย์*
Regulator	4	10	62.50 %
Buck	7	16	32.70 %
OR Gate	18	73	22.50 %
Phase-Splitting	33	125	11.50 %
Triangular Wave Generator	48	268	11.63%
RC_3	3	7	77.80 %
RC_5	5	13	52.00 %
RC_{10}	10	28	28.00 %
RC_{30}	30	88	9.78 %
RC_{50}	50	148	5.92 %
RC_{100}	100	298	2.98 %

$$\begin{array}{c}\text{* ดัชนีมากเลขศูนย์} \\ \text{(Sparsity Index)}\end{array} = \frac{\text{จำนวนสมาชิกที่มีค่าไม่เป็นศูนย์}}{\text{จำนวนสมาชิกทั้งหมดในเมทริกซ์}} \times 100\%$$

ตารางที่ ก.1 รายละเอียดของเมทริกซ์สัมประสิทธิ์ของสมการวงจรทดสอบ

วงจรทดสอบแต่ละวงจร มีรายละเอียดดังนี้

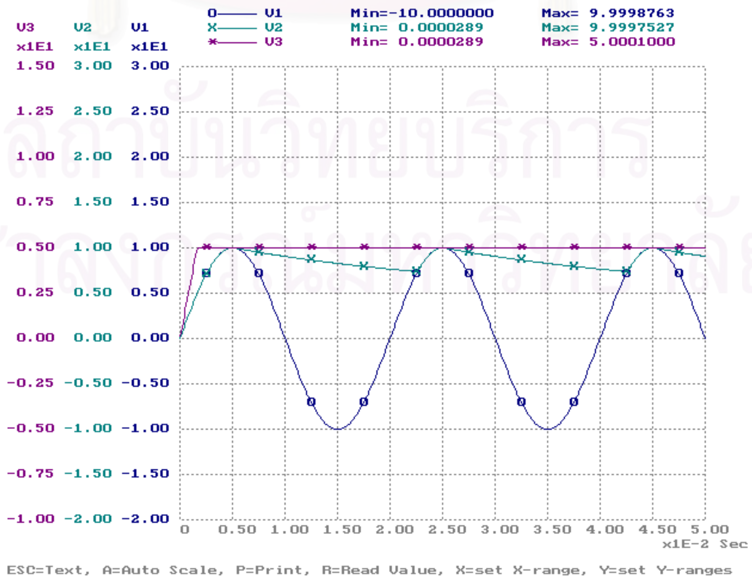
1. วงจร Regulator



รูปที่ ก.1 วงจร Regulator

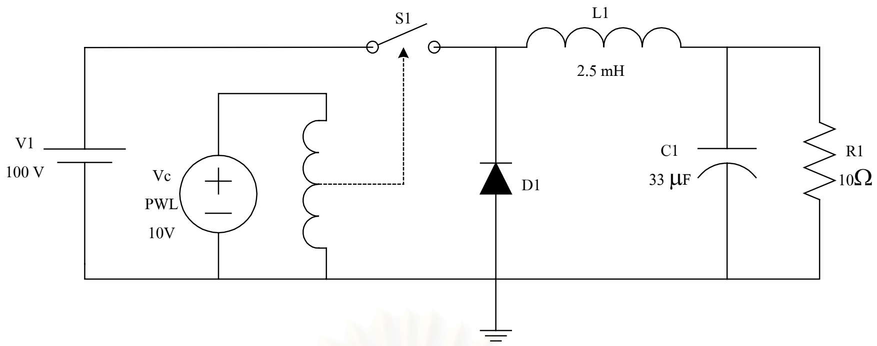
วงจร Regulator ในรูปที่ ก.1 เป็นวงจรแปลงสัญญาไฟฟ้ากระแสสลับให้เป็นกระแสตรง โดยการป้อนข้อมูลตามข้อความข้างล่างนี้ให้โปรแกรม LEK จะได้ผลการจำลองวงจรแสดงในรูปที่ ก.2

```
V1  1  0  Rs=1mOhm  Sine  Vm=10  f=50  Ph=0Degree ;
D1  1  2  #1N916  Vd=0.6 ;
#1n916  D  PWL  Vcutin=0.0V  Ron=1mOhm   Roff=100Meg  Vz=1Meg  Rz=1mOhm ;
C2  2  0  470uF  VC(0)=0.0Volt ;
Rs  2  3  50 ;
Dz  0  3  #Zener  Vd=0.6 ;
#Zener  D  PWL  Vcutin=0.0V  Ron=1mOhm Roff=100Meg Vz=5Volt Rz=1mOhm ;
\  T  1=V1  2=V2  3=V3  Tstop=50mS  Tstep=0.1mS StepCtrl=FIX  MaxTRiter=15 ;
```



รูปที่ ก.2 ผลการจำลองการทำงานของวงจร Regulator

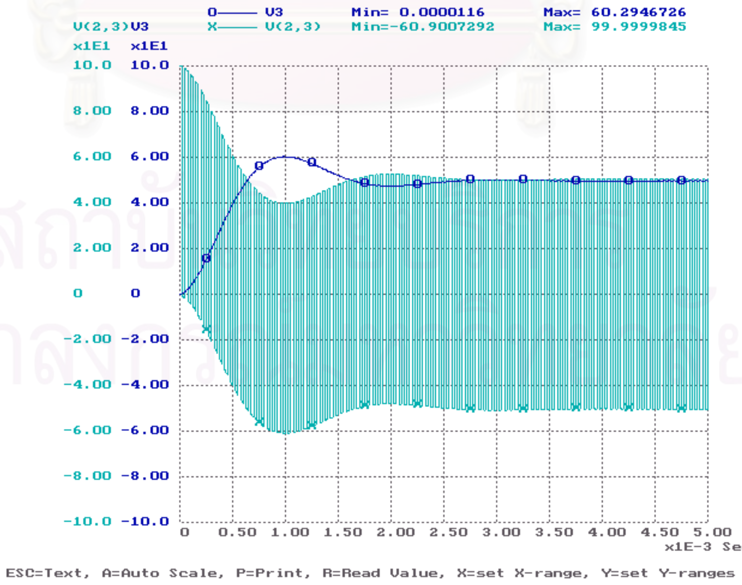
2. วงจร Buck



รูปที่ ก.3 วงจร Buck

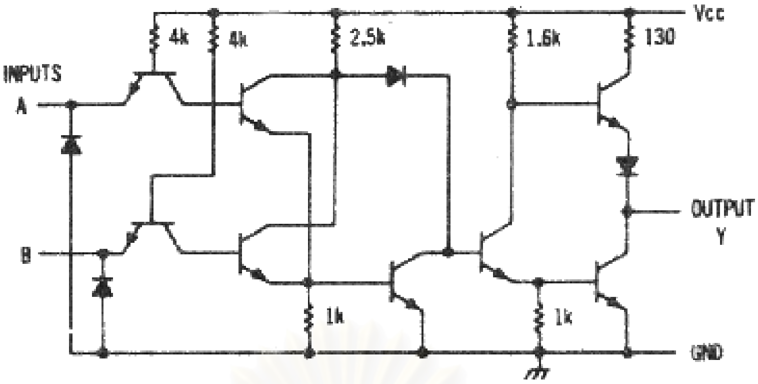
วงจร Buck ในรูปที่ ก.3 เป็นวงจรลดทอนแรงดันไฟฟ้า โดยการป้อนข้อมูลตามข้อความข้างล่างนี้ให้โปรแกรม LEK จะได้ผลการจำลองวงจรแสดงในรูปที่ ก.4

```
Vs  1  0  Rs=0  Dc  100Volts ;
D1  0  2  #1N916  Vd=0.6 ;
L1  2  3  2.5mH  IL(0)=0.0A ;
C1  3  0  33uF  VC(0)=0.0V ;
R1  3  0  10Ohms ;
Vc  4  0  Rs=0  PWL  #BrkPts=5  T1=0S  V1=0Volt  T2=0S  V2=10Volts  T3=25uS  V3=10Volts
    T4=25uS  V4=0Volt  T5=50uS  V5=0Volt ;
S1  NO  1  2  Rs=1mOhm  4  0  Vth=5Volts  Vdelta=0  Rcoil=1e10 ;
#1N916  D  PWL  Vcutin=0.6V  Ron=1mOhm  Roff=100Meg  Vz=1Meg  Rz=1mOhm ;
\  T  Vplot1=V3  Vplot2=V(2,3)  Vplot3=  Tstop=50ms  Tstep=0.1ms  StepCtrl=Adj  LTEv=0.1V
    LTEi=0.1A  MaxTRiter=15 ;
```



รูปที่ ก.4 ผลการจำลองการทำงานของวงจร Buck

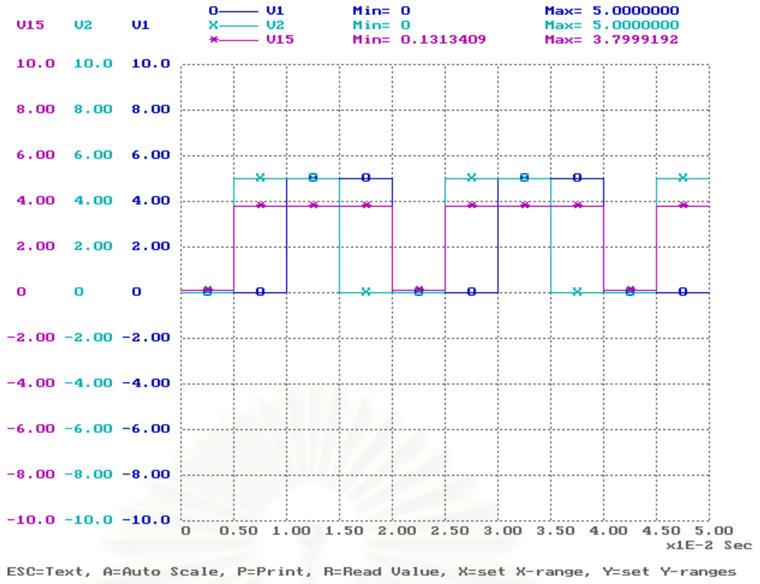
3.วงจร OR Gate



รูปที่ ก.5 วงจร OR Gate

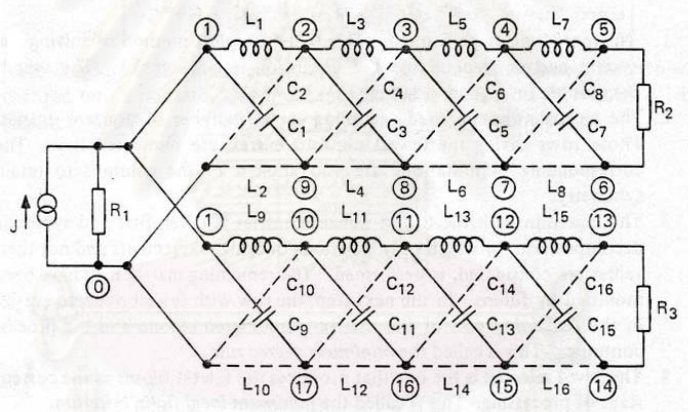
วงจร OR Gate ในรูปที่ ก.5 เป็นวงจรสำหรับคำนวณคำสั่ง OR ใน IC เบอร์ 7432 (Quadruple 2-Input Positive-OR Gate)[18] โดยการป้อนข้อมูลตามข้อความข้างล่างนี้ให้โปรแกรม LEK จะได้ผลการจำลองวงจรแสดงในรูปที่ ก.6

```
#BC549  Q  NPN PWL  BetaF=100  BetaR=1  Cbc=0  Cbe=0  Vcutin=0.6V  Rpi=1K  Rce=100Meg ;
#1N916  D  PWL  Vcutin=0.6V  Ron=1mOhm  Roff=100Meg  Vz=1Meg  Rz=1mOhm ;

VA  1  0  Rs=0  Pwl  #BrkPts=4  T1=0.0  V1=0  T2=0.01  V2=0  T3=0.01  V3=5  T4=0.02  V4=5 ;
VB  2  0  Rs=0  Pwl  #BrkPts=6  T1=0.0  V1=0  T2=0.005  V2=0  T3=0.005  V3=5  T4=0.015  V4=5
     T5=0.015  V5=0  T6=0.02  V6=0 ;
Vcc  3  0  Rs=0  DC  5 ;

D1  0  1  #1N916  Vd=0.6 ;
D2  0  2  #1N916  Vd=0.6 ;
Q1  6  4  1  #BC549  VBC=-1V  VBE=0.6 ;
R1  3  4  4k ;
Q2  7  5  2  #BC549  VBC=-1V  VBE=0.6 ;
R2  3  5  4k ;
Q3  8  6  9  #BC549  VBC=-1V  VBE=0.6 ;
Q4  8  7  9  #BC549  VBC=-1V  VBE=0.6 ;
R3  3  8  2.5k ;
R4  9  0  1k ;
D3  8  10  #1N916  Vd=0.6 ;
Q5  10  9  0  #BC549  VBC=-1V  VBE=0.6 ;
Q6  11  10  12  #BC549  VBC=-1V  VBE=0.6 ;
R5  3  11  1.6k ;
R6  12  0  1k ;
R7  3  13  130 ;
Q7  13  11  14  #BC549  VBC=-1V  VBE=0.6 ;
D4  14  15  #1N916  Vd=0.6 ;
Q8  15  12  0  #BC549  VBC=-1V  VBE=0.6 ;
\ T  Vplot1=V1  Vplot2=V2  Vplot3=V15  Tstop=50ms  Tstep=0.1ms  StepCtrl=FIX  MaxTRiter=50 ;
```



รูปที่ ก.6 ผลการจำลองการทำงานของวงจร OR Gate

4. วงจร Phase-Splitting

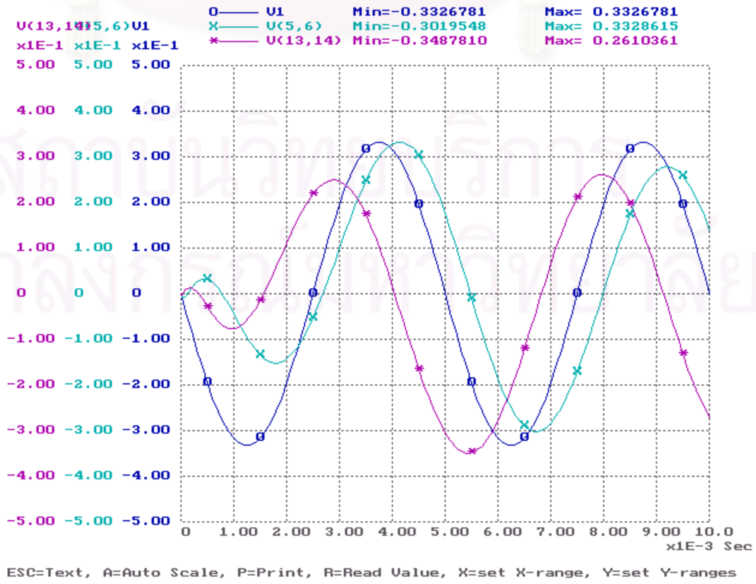


รูปที่ ก.7 วงจร Phase-Splitting

วงจร Phase-Splitting ในรูปที่ ก.7 เป็นวงจรแยกสัญญาณไฟฟ้าออกเป็น 2 สัญญาณที่มีเฟสต่างกัน 90° โดยนำมาจาก [19] โดยการป้อนข้อมูลตามข้อความข้างล่างนี้ให้โปรแกรม LEK จะได้ผลการจำลองวงจรแสดงในรูปที่ ก.8

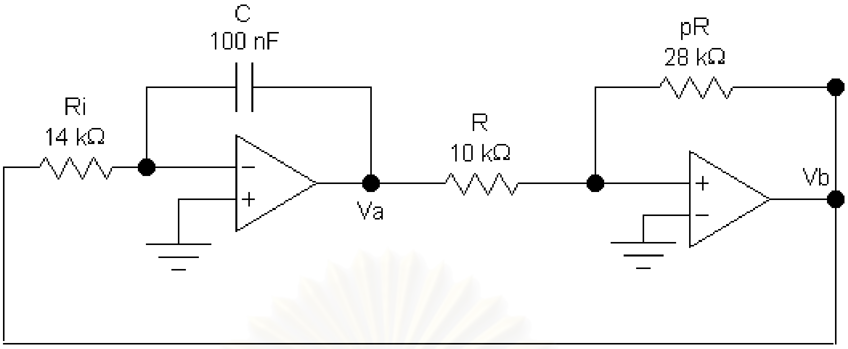
```
I1  1  0  Gs=0siemen  Sine  Im=1  f=200  Ph=0Degree ;
R1  1  0  1 ;

L1  1  2  0.102888m  IL(0)=0.0A ;
L2  0  9  0.102888m  IL(0)=0.0A ;
C1  1  9  0.102888m  VC(0)=0.0V ;
C2  2  0  0.102888m  VC(0)=0.0V ;

L3  2  3  0.493852m  IL(0)=0.0A ;
L4  9  8  0.493852m  IL(0)=0.0A ;
```

```
C3  2  8  0.493852m  VC(0)=0.0V ;
C4  9  3  0.493852m  VC(0)=0.0V ;

L5  3  4  2.15717m  IL(0)=0.0A ;
L6  8  7  2.15717m  IL(0)=0.0A ;
C5  3  7  2.15717m  VC(0)=0.0V ;
C6  8  4  2.15717m  VC(0)=0.0V ;

L7  4  5  17.2722m  IL(0)=0.0A ;
L8  7  6  17.2722m  IL(0)=0.0A ;
C7  4  6  17.2722m  VC(0)=0.0V ;
C8  7  5  17.2722m  VC(0)=0.0V ;
R2  5  6  1 ;

L9   1  10  0.0293307m  IL(0)=0.0A ;
L10  0  17  0.0293307m  IL(0)=0.0A ;
C9   1  17  0.0293307m  VC(0)=0.0V ;
C10  0  10  0.0293307m  VC(0)=0.0V ;

L11  10  11  4.87647m  IL(0)=0.0A ;
L12  17  16  4.87647m  IL(0)=0.0A ;
C11  10  16  4.87647m  VC(0)=0.0V ;
C12  17  11  4.87647m  VC(0)=0.0V ;

L13  11  12  1.02583m  IL(0)=0.0A ;
L14  16  15  1.02583m  IL(0)=0.0A ;
C13  11  15  1.02583m  VC(0)=0.0V ;
C14  16  12  1.02583m  VC(0)=0.0V ;

L15  12  13  0.234847m  IL(0)=0.0A ;
L16  15  14  0.234847m  IL(0)=0.0A ;
C15  12  14  0.234847m  VC(0)=0.0V ;
C16  15  13  0.234847m  VC(0)=0.0V ;
R3  13  14  1 ;
\   T   Vplot1=V1   Vplot2=V(5,6)   Vplot3=V(13,14)   Tstop=50ms   Tstep=0.1ms   StepCtrl=FIX
    MaxTRiter=50 ;
```

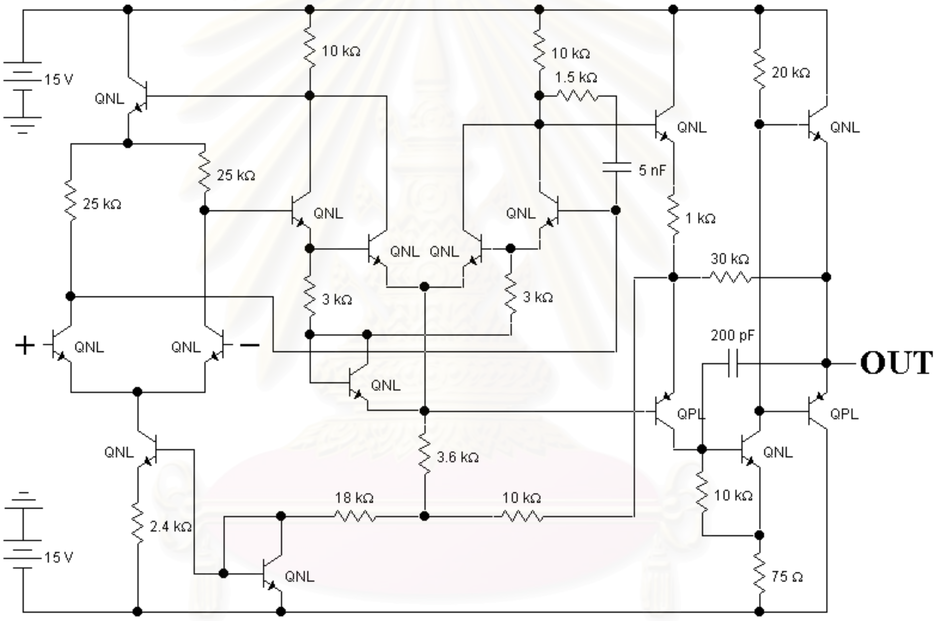


รูปที่ ก.8 ผลการจำลองการทำงานของวงจร Phase-Splitting

5. วงจร Triangular Wave Generator



รูปที่ ก.9 วงจร Triangular Wave Generator



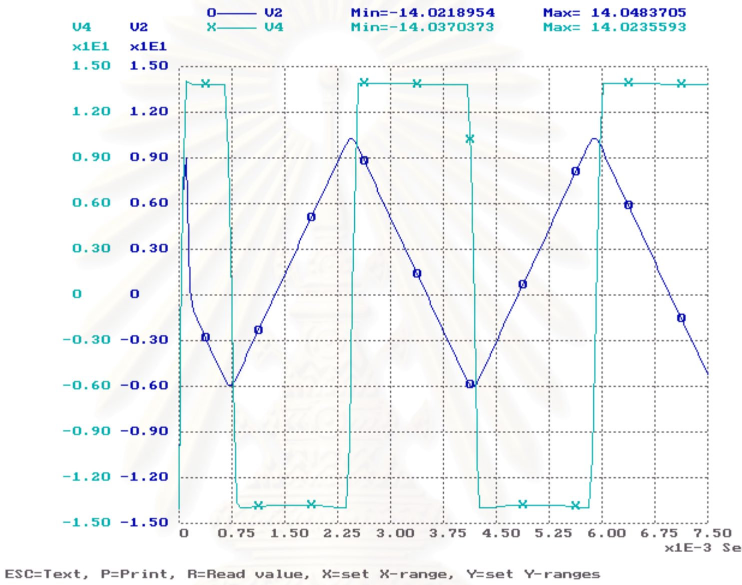
รูปที่ ก.10 ส่วนของวงจร Op-amp ในวงจร Triangular Wave Generator

วงจร Triangular Wave Generator เป็นวงจรรูปคลื่นสามเหลี่ยน ซึ่งประกอบด้วย Op-amp เบอร์ μA709 ในรูปที่ ก.10 จำนวน 2 ตัวมาต่อกันดังในรูปที่ ก.9 โดยการป้อนข้อมูลตามข้อความข้างล่างนี้ให้โปรแกรม LEK จะได้ผลการจำลองวงจรแสดงในรูปที่ ก.11

```
V0 17 0 Rs=0 DC 15V
V1 0 23 Rs=0 DC 15V
R0 39 32 3Kohm
R1 17 38 10Kohm
R2 38 35 1.5Kohm
R3 46 30 1Kohm
R4 17 41 20Kohm
R5 17 36 10Kohm
R6 37 27 25Kohm
```

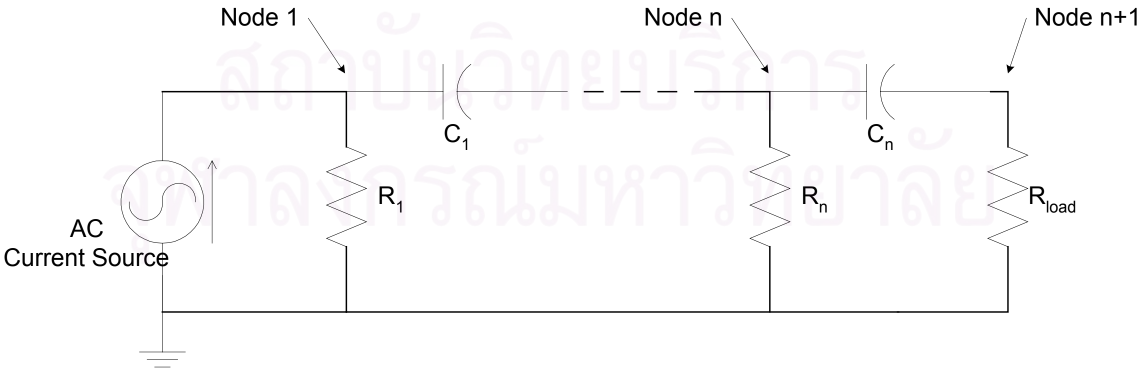
```
R7 37 45 25Kohm
R8 42 29 18Kohm
R9 44 0 1Kohm
R10 43 23 2.4Kohm
R11 33 29 3.6Kohm
R12 30 2 30Kohm
R13 40 23 75ohm
R14 34 40 10Kohm
R15 31 32 3Kohm
R16 29 30 10Kohm
R17 20 12 3Kohm
R18 17 19 10Kohm
R19 19 15 1.5Kohm
R20 26 10 1Kohm
R21 17 22 20Kohm
R22 17 16 10Kohm
R23 18 8 25Kohm
R24 18 6 25Kohm
R25 24 9 18Kohm
R26 25 23 2.4Kohm
R27 13 9 3.6Kohm
R28 10 4 30Kohm
R29 21 23 75ohm
R30 14 21 10Kohm
R31 11 12 3Kohm
R32 9 10 10Kohm
R33 2 5 10Kohm
R34 5 4 28Kohm
R35 4 1 14Kohm
R36 3 0 1Kohm

C0 34 2 200pF VC(0)=0V
C1 35 27 5nF VC(0)=0V
C2 14 4 200pF VC(0)=0V
C3 15 8 5nF VC(0)=0V
C4 1 2 100nF VC(0)=0V

Q0 38 39 33 #Qnpn VBC=0 VBE=0
Q1 17 38 46 #Qnpn VBC=0 VBE=0
Q2 17 41 2 #Qnpn VBC=0 VBE=0
Q3 36 45 31 #Qnpn VBC=0 VBE=0
Q4 36 31 33 #Qnew VBC=0 VBE=0
Q5 45 44 28 #Qnpn VBC=0 VBE=0
Q6 28 42 43 #Qnpn VBC=0 VBE=0
Q7 42 42 23 #Qnpn VBC=0 VBE=0
Q8 32 32 33 #Qnpn VBC=0 VBE=0
Q9 41 34 40 #Qnpn VBC=0 VBE=0
Q10 38 27 39 #Qnpn VBC=0 VBE=0
Q11 17 36 37 #Qnpn VBC=0 VBE=0
Q12 27 1 28 #Qnpn VBC=0 VBE=0
Q13 19 20 13 #Qnpn VBC=0 VBE=0
Q14 17 19 26 #Qnpn VBC=0 VBE=0
Q15 17 22 4 #Qnpn VBC=0 VBE=0
Q16 16 6 11 #Qnpn VBC=0 VBE=0
Q17 16 11 13 #Qnew VBC=0 VBE=0
Q18 7 24 25 #Qnpn VBC=0 VBE=0
Q19 24 24 23 #Qnpn VBC=0 VBE=0
Q20 12 12 13 #Qnpn VBC=0 VBE=0
Q21 22 14 21 #Qnpn VBC=0 VBE=0
Q22 19 8 20 #Qnpn VBC=0 VBE=0
```

```
Q23 17 16 18 #Qnpn VBC=0 VBE=0
Q24 8 3 7 #Qnpn VBC=0 VBE=0
Q25 6 5 7 #Qnpn VBC=0 VBE=0

Q26 23 41 2 #Qpnp VBC=0 VBE=0
Q27 34 33 30 #Qpnp VBC=0 VBE=0
Q28 23 22 4 #Qpnp VBC=0 VBE=0
Q29 14 13 10 #Qpnp VBC=0 VBE=0

#Qnpn Q NPN EXP  BF=80 BR=960m Cbe=3pF Cbc=2pF Is=1e-16AVA=50V
#Qnew Q NPN EXP BF=80 BR=960m Cbe=3pF Cbc=2pF Is=1e-16AVA=50V
#Qpnp Q PNP EXP BF=10 BR=960m Cbe=6pF Cbc=4pF Is=1e-16AVA=50V
\ T  Vplot1=V2  Vplot2=V4  Vplot3=  Tstop=10ms  Tstep=0.1ms  StepCtrl=FIX  MaxTRiter=50 ;
```



รูปที่ ก.11 ผลการจำลองการทำงานของวงจร Triangular Wave Generator

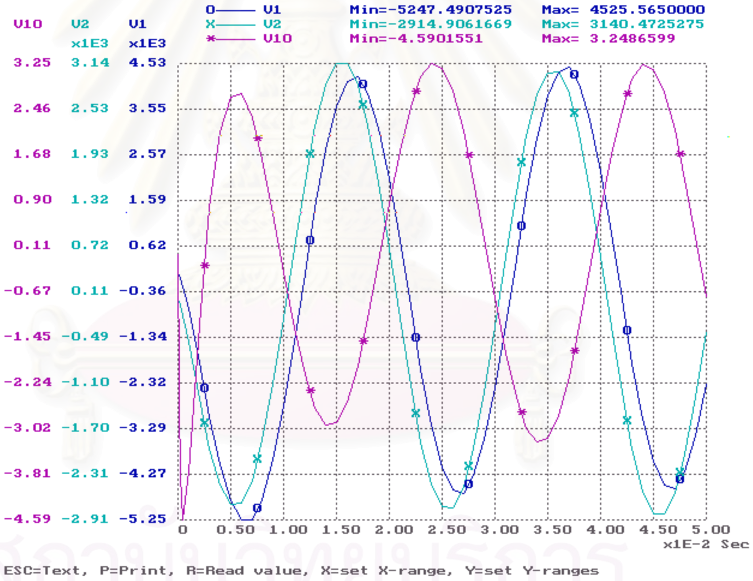
6. วงจร RC Ladder



รูปที่ ก.12 วงจร RC Ladder

วงจร RC Ladder ที่นำมาทดสอบมี 6 วงจร โดยมีจำนวนปมในวงจร (n) เป็น 3, 5, 10, 30, 50 และ 100 ปม รูปวงจรแสดงดังรูปที่ ก.12 ข้อมูลที่ป้อนให้โปรแกรมจะมีขนาดขึ้นอยู่กับจำนวนปมในวงจรนั้นๆ และผลการจำลองวงจรที่ได้จะมีรูปร่างคล้ายในรูปที่ ก.13

```
I  1  0  Gs=0siemen  Sine  Im=1  f=50  Ph=0Degree
R1  1  0  10k ;
C1  1  2  1E-6  VC(0)=0 ;
R2  2  0  10k ;
C2  2  3  1E-6  VC(0)=0 ;
R3  3  0  10k ;
C3  3  4  1E-6  VC(0)=0 ;
…
…
…
…
R[n-1]  [n-1]  0  10k ;
C[n-1]  [n-1]  [n]  1E-6  VC(0)=0 ;
Rl  [n]  0  100 ;
\  T  Vplot1=V1  Vplot2=V2  Vplot3=V[n]  Tstop=1s  Tstep=1mS  StepCtrl=FIX  MaxTRiter=15 ;
```



รูปที่ ก.13 ผลการจำลองการทำงานของวงจร RC Ladder

ภาคผนวก ข

บทความที่ได้รับการตีพิมพ์

บทความที่ได้รับการตีพิมพ์

งานวิจัยในส่วนการสร้างสูตรสำเร็จการแก้สมการเมทริกซ์ด้วยรหัสคำสั่งเครื่อง ได้รับการตีพิมพ์เป็นบทความวิชาการในการประชุมวิชาการ “The 2000 IEEE Asia-Pacific Conference on Circuits and Systems” หรือ APCCAS 2000 ซึ่งจัดขึ้นที่ Crystal Palace Hotel เมืองเทียนจิน ประเทศสาธารณรัฐประชาชนจีน ระหว่างวันที่ 4 ถึง 7 ธันวาคม 2543 บทความดังกล่าวผู้วิจัยและท่านอาจารย์ที่ปรึกษาวิทยานิพนธ์ได้ร่วมกันเขียนขึ้นโดยมีเนื้อหาดังนี้

ภาคผนวก ค

รายละเอียดของโปรแกรมส่วนเมทริกซ์มากเลขศูนย์

Machine Code Generating to Speed-up the circuit simulation

Noppadol Chitcharus and Ekachai Leelarasmee
Chulalongkorn University, Electrical Engineering Department
Phya-Thai Road, Patumwan, Bangkok 10330 Thailand.
Phone: (662)218-6488, E-mail address: b0355521@student.chula.ac.th

Abstract

Machine code generating is a technique that generates a sequence of machine codes for solving a special circuit equation. The equation solving operations differ for various circuits due to their different sparse matrix structure. Hence, the machine codes must be generated in the run-time preprocessing. The machine codes will be the shortest instructions for solving the circuit equation without any looping or comparision and ready for excution by CPU. This machine code generating technique can speed-up the circuit simulation program from 1 to 5 times.

I. Introduction

Most circuit simulation programs calculate values of the circuit variables by using numerical methods and process as presented in fig.1. The main process is in solving a matrix circuit equation (1) formulated by the modified nodal analysis method[1].

$$Ax = b \qquad (1)$$

where A is an n × n coefficient matrix, b is an n-vector of known constants, x is an n-vector of circuit variables, and n is the number of circuit variables.

The equation solver spends more than 70% of the whole circuit simulation time on solving the same circuit equation for many thousand times. Therefore if the code generating technique[2][3] is applied in the simulation process as presented in fig.2, the execution time can be reduced significantly because the operation codes, generated in the preprocessing step, give the shortest calculation steps and contain no looping or comparison.

The operation codes can be generated in several formats such as the interpretive code[3]. Anyhow the interpretive code requires some execution time to interpret the codes, so we decide to generate codes in a format of machine code that CPU can operate immediately for fastest matrix equation solving.

II. Useful benefit

The advantage of machine code generating technique is in its rapid code execution because the codes have a linear structure, contain only necessary operations to solve the matrix equation and eliminate unnecessary operations such as DO, IF, GOTO, or other logic statements. Furthermore, the calculation by machine codes can use the registers of CPU as the temporary variables during the calculation in each step. Hence there are less data transfer between CPU and external memory.

In this paper, The machine codes are generated specially for the CPU in the Pentium Processor Family. The floating-point instructions[5], their initial letter is "F", can be grouped into six functional categories : Data transfer instructions, Basic arithmetic instructions, Comparison instructions, Transcendental instructions, Load constant instructions and FPU control instructions.

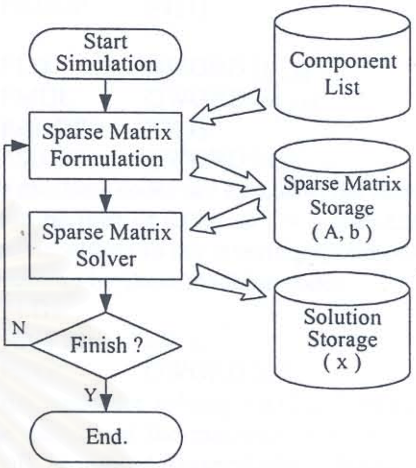


Fig.1 General process of circuit simulator.

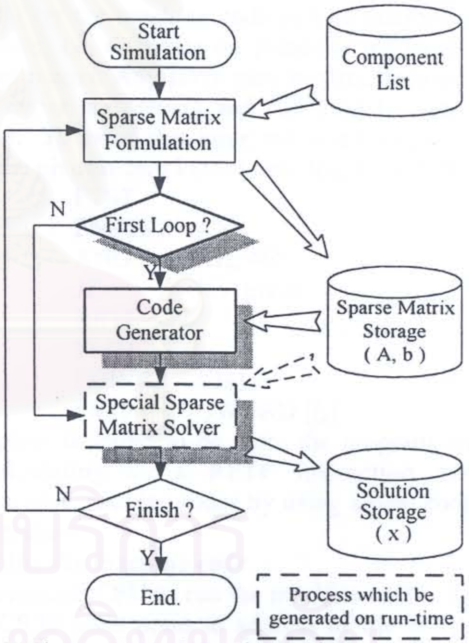


Fig.2 The process of circuit simulation with code generating technique.

The instructions that are used in the machine codes for matrix equation solving are :

Data Transfer Instructions:

FLD	Load Real
FSTP	Store Real and Pop

Load Constants Instruction:

FLD1	Load +1.0

Basic Arithmetic Instructions:

FADD / FADDP	Add real
FSUBR	Reverse subtract real
FMUL	Multiply real
FDIVR / FDIVRP	Reverse divide

The real numbers are stored in 8 bytes in memory and the instructions that will operate with this real number will be represented as

```
Fxxx     QWORD [ offset ]
```

This means that the real number is taken from address [DS:offset] to [DS:offset+7] to operate with the value in the register of CPU.

III. Machine Code Generating

1. Matrix equation solving by LU factorization

Many circuit simulation programs solve the matrix circuit equation by LU factorization method[5]. By this algorithm, The coefficient matrix A in Eq. (1) must be factored into a lower triangular matrix L and U, a upper triangular matrix whose diagonal elements are 1, by Crout's algorithm as follows:

$$l_{ik} = a_{ik} - \sum_{m=1}^{k-1} l_{im} u_{mk} \; ; \qquad i \geq k \; (2)$$

$$u_{kj} = \left(a_{kj} - \sum_{m=1}^{k-1} l_{km} u_{mj} \right) \Big/ l_{kk} \; ; \qquad j > k \; (3)$$

So, Eq. (1) can be rewritten as $LUx = b$ and by defining an auxiliary vector $z = Ux$, it can be calculated from

$$z_i = \left(b_i - \sum_{j=1}^{i-1} l_{ij} z_j \right) \Big/ l_{ii} \; ; \qquad i = 1,2,\ldots,n \; (4)$$

Finally, a vector x can be calculated from

$$x_i = z_i - \sum_{j=i+1}^{n} u_{ij} z_j \; ; \qquad i = n,n-1,\ldots,1 \; (5)$$

In actual program, only non-zero elements are stored and coded for operations. This is well known as sparse matrix techniques[6].

2. Inverting l_{ii}

Because the elements on the principal diagonal in the matrix L, i.e. l_{ii}, will only be used to divide other elements, we can store the value of $1/l_{ii}$ in the matrix L instead of l_{ii}. Hence the division by l_{ii} will be changed to multiplication by its reciprocal.

Changing from division to multiplication can reduce the calculation time because a division instruction takes more execution time than a multiplication instruction. On a Pentium Processor, an FDIV instruction takes 33 clocks but an FMUL instruction takes only 3 clocks.

3. Machine code format

The format of machine codes that are generated for calculating l_{ik} in Eq. (2) is presented as

FLD QWORD [l_{i1}]
FMUL QWORD [u_{1k}]
FLD QWORD [l_{i2}]
FMUL QWORD [u_{2k}]
FADDP ST(1)
...
FLD QWORD [$l_{i\,k-1}$]
FMUL QWORD [$u_{k-1\,k}$]
FADDP ST(1)
FSUBR QWORD [a_{ik}]

After these machine codes are executed, the result of l_{ik} will be stored in register of CPU. If $i = k$ then the result will be inverted according to the inverting l_{ii} technique and stored in the memory by these machine codes :

FLD1
FDIVRP ST(1)
FSTP QWORD [l_{ii}]

The machine codes for solving Eq.(3),(4) and (5) will have the same pattern as the machine codes for solving Eq.(2) and will be stored respectively. Finally, these machine codes must end with a RETF (Return Far) instruction as the last code.

Solving by the machine code can be guarded against small pivots, i.e. elements on principle diagonal of the coefficient matrix. Any pivot may become zero while the machine codes are solving and "division by zero error" will occur. To avoid this error, we insert a few codes to check if the pivot is zero before inverting l_{ii}, as follows:

FTST
FNSTSW AX
AND AH, 40h
JZ Inv_pivot
RETF
Inv_pivot : FLD1
FDIVRP ST(1)
FSTP QWORD [l_{ii}]

If a pivot is found to be zero, the program will exit from calculating by a RETF Instruction and will regenerate new machine codes by using a new coefficient matrix structure.

4. Running the machine code

To command CPU to run the machine codes, the CPU register CS:IP must point to address of the array which stores the machine codes by a CALL FAR instruction as follows:

```
PUSH      ES
PUSH      DS
MOV       ES, [Segment of b]
MOV       DS, [Segment of A]
CALL FAR [Address of MachineCode]
POP       DS
POP       ES
```

Registers DS and ES are initialized to be the segment of the coefficient matrix *A* and the constant vector *b* respectively because data of matrix *A* and *b* are assumed to be in different segments of memory. Usually, a machine code cans only access to data in the segment DS. Therefore when data in the vector *b* must be read or written, a 26h code must be added in front of each instruction to signify the CPU that this instruction will access data in the ES segment.

Assigning ES as the segment of matrix *b* because data in matrix *b* is less accessed then data in matrix *A*, then spend lesser bytes to store the 26h codes in machine codes.

IV. Mathematical comparison

According to the machine code format, there is no looping or comparison between solving. Furthermore, this technique can use the registers of CPU as the temporary variables during calculating in each step. Hence there are less data transfer between the CPU and its external memory. Table 1 shows the comparison of number of executions for both methods.

Execution	Number of Executions	
	Normal Method	Machine Code
Looping	$\frac{1}{3}n^3+n^2+\frac{8}{3}n+1$	0
Floating-Point Loading	$\frac{1}{3}n^3+n^2+\frac{2}{3}n$	$\frac{3}{2}n^2+\frac{1}{2}n$
Floating-Point Storing	$\frac{1}{3}n^3+n^2+\frac{2}{3}n$	n^2+n
Comparison	$\frac{1}{3}n^3+\frac{1}{2}n^2+\frac{1}{6}n$	n
Addition and Subtraction	$\frac{1}{3}n^3+\frac{3}{2}n^2-\frac{5}{6}n$	Same
Multiplication	$\frac{1}{3}n^3+n^2-\frac{1}{3}n$	Same
Division	n	Same

* n = Number of circuit variables

Table 1 Comparison of the number of executions between normal sparse matrix solving method and solving by machine code (Full matrix condition).

V. The Experimental Results

LEK[7] version 6.0S is the circuit simulator program running under the MS-DOS operating system and uses the modified nodal analysis and the sparse matrix techniques. The new version, LEK 6.0M, was developed from LEK 6.0S by including the machine code generating technique. Both programs were tested and their results were compared to verify the efficiency of machine code generating technique.

Both programs simulated 11 test circuits on a Pentium 100-MHz PC. Solving time and size of memory, which stored sparse matrix and machine codes, are measured and tabulated in Table 2.

The 11 test circuits consist of 5 practical circuits (Regulator, Buck Converter, OR gate, Phase-splitting and Triangular wave generator circuit) and 6 RC-Ladders circuits that have 3, 5, 10, 30 and 100 circuit variables respectively. All of test circuits are showed in fig.3 to fig.8.

Circuit	n	Time (Ticks*)		Time Ratio	Sparse Matrix (Bytes)	Machine Code (Bytes)
		LEK 6.0S	LEK 6.0M			
Regulator	4	4	2	2.00	120	312
Buck	7	272	130	2.09	192	488
OR Gate	18	50	17	2.94	1,248	3,398
Phase-Splitting	33	69	14	4.93	2,142	5,796
Triangular wave generator	48	934	248	3.77	9,540	18,106
RC3	3	6	4	1.50	84	222
RC5	5	11	5	2.20	156	402
RC10	10	21	9	2.33	336	852
RC30	30	81	29	2.79	1,056	2,652
RC50	50	168	47	3.57	1,776	4,452
RC100	100	507	98	5.17	3,576	8,952

* 1 Tick = 1 / 18.2 second

Table 2 The Experimental Results.

The experimental results show that machine code generating technique can reduce the computation time about 1 to 5 times but use memory to store machine codes about 2 to 3 times that of memory to store sparse matrix data. The reducing time and increasing memory usage depend on characteristic and size of each circuit.

In case of RC-Ladder circuits, computation time of LEK 6.0S is superlinearly dependent on the number of circuit variables (*n*). But the computation time of LEK 6.0M linearly depend on *n* while machine codes's size also linearly depends on *n*.

VI. Conclusions

Machine code generating technique can speed-up the circuit simulation program more than 1 to 5 times depend on the simulated circuit because machine codes consist of only necessary calculating for matrix equation solving and can use the registers in CPU as temporary variables. But the increasing in memory usage must be in consideration.

References

[1] C. W. Ho, A. E. Ruehli, and P. A. Brennan, "The Modified Nodal Approach to Network Analysis", IEEE Trans. on Circuits and Systems, Vol. CAS-22, June 1975, pp.504-509.

[2] Jan Ogrodzki, Circuit Simulation Methods and Algorithms, CRC Press : Florida, 1994.

[3] Jiri Vlach, and Kishore Singhal, Computer Methods for Circuit Analysis and Design, Van Nostrand Reinhold : New York, 1983.

[4] Intel Corporation, Intel Architecture Software Developer's Manual. vol. 1, 1999.

[5] L. O. Chua, and P. M. Lin, Computer Aided Analysis of Electronic Circuits: Algorithms and Computational Techniques, Prentice Hall Inc. : New Jersey, 1975.

[6] K. S. Kundert, Sparse Matrix Techniques. In:Circuit Analysis Simulation and Design. Vol. 1., edited by A.E. Ruehli, Elsevier : North Holland, 1986.

[7] Ekachai Leelarasmee, STDB Designed RD&E Project Final Report, Chulalongkorn University : Bangkok, 1991.

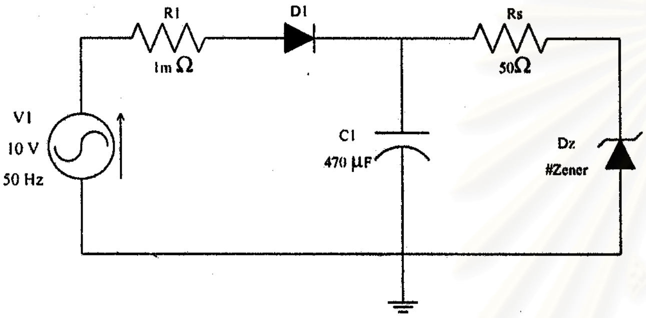


Fig. 3 Regulator circuit.

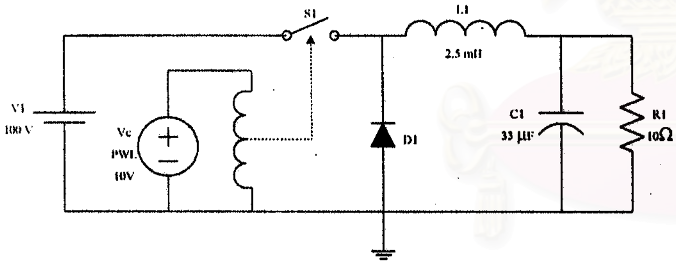


Fig. 4 Buck converter circuit.

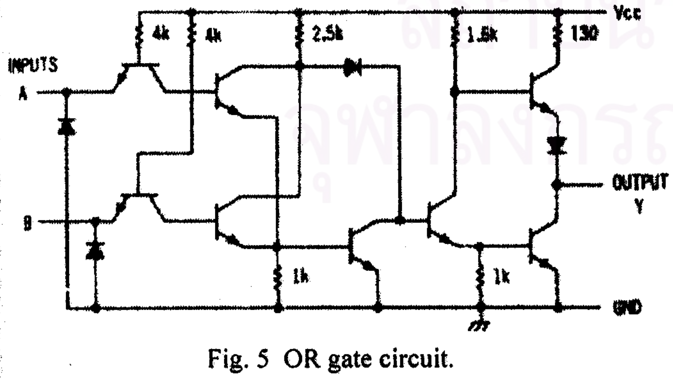


Fig. 5 OR gate circuit.

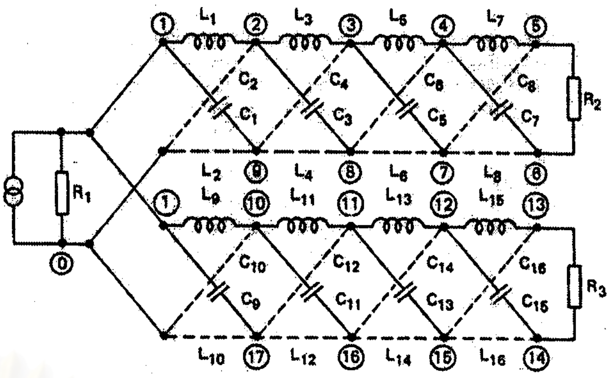


Fig. 6 Phase-Splitting circuit.

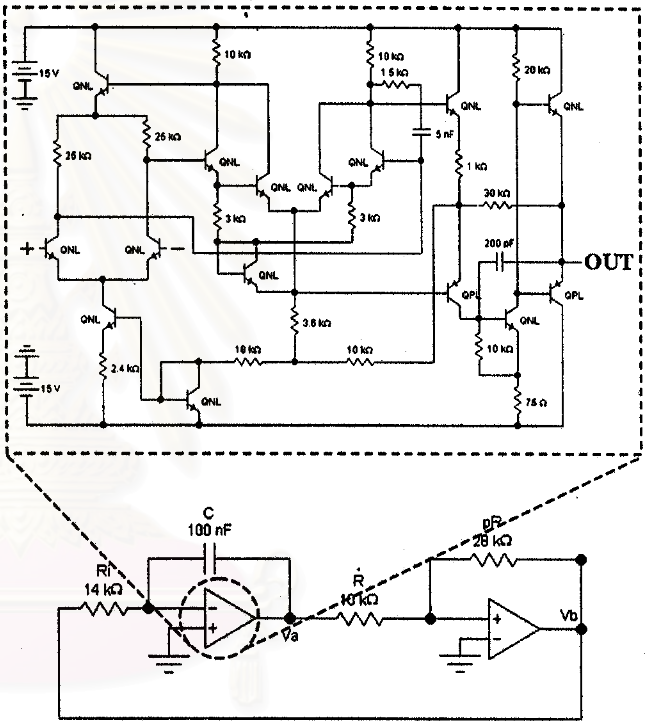


Fig. 7 Triangular wave generator.

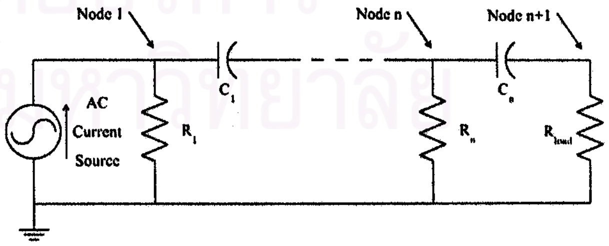


Fig. 8 RC-Ladder circuit.

ภาคผนวก ค

รายละเอียดของโปรแกรมส่วนเมทริกซ์มากเลขศูนย์

รายละเอียดของโปรแกรมส่วนเมทริกซ์มากเลขศูนย์

งานวิทยานิพนธ์นี้ได้ดัดแปลงโปรแกรม “เล็ก 6.0” โดยเปลี่ยนขั้นตอน Load Matrix Equation, LU-Factor และ LU-Solve ให้เป็นขั้นตอน Sparse Load Matrix Equation, Sparse LU-Factor และ Spare LU-Solve ตามลำดับ ขั้นตอนสำคัญทั้ง 3 ส่วนนี้ถูกบรรจุอยู่ใน Unit SparseU นอกจากในโปรแกรมส่วนนี้ได้รวมส่วนการแก้สมการด้วยรหัสคำสั่งด้วย โดยมีรายละเอียดดังนี้

```
Unit SparseU;

Interface

Uses  Common;

Const NearZero = 1E-35;
     MaxIndex = 65535;
     SizePerOpCode = 1;

Type  ValueArrType = Array[1..30] of Float;
     ValueArrP    = ^ValueArrType;
     IndexType    = Word;
     IndexArrType = Array[1..30] of IndexType;
     IndexArrP    = ^IndexArrType;
     RCArrType    = Array[0..30] of IndexType;
     RCArrP       = ^RCArrType;

Var   ValueP : ValueArrP;
    { ValueP = Values in the Matrix }

     SparseSize, Dimension : IndexType;
    { SparseSize = Size of Sparse matrix }
    { Dimension = Dimension of the Matrix }

     NewC : IndexArrP;
    { New Ordering of column after reordering }

     OpCode_LU_Solve : IndexArrP;
     OpCode_LU_Solve_Size,
     OpCode_LU_Solve_Length : IndexType;

     StartInst : Array[1..6] of Byte;
   { To use Opcode , you must store }
   { ValueP, SparseSize, Dimension, NewC }
   { OpCode_LU_Solve, OpCode_LU_Solve_Size, OpCode_LU_Solve_Lenght }

     NewR : IndexArrP;
    { New Ordering of row after reordering }
     FloatOverFlow : Boolean; { Floating Point Over Flow Check }
     UseOpcode : Boolean;    { Use Opcode to solve the equation }

Procedure Init_Sparse(d:IndexType);
Procedure Clear_Sparse;
Procedure Destroy_Sparse;
Procedure Input_Data(V:Float; R,C:IndexType);
Function  Read_Data(R,C:IndexType) :Float;
Function Finish_Input_Data : Boolean;
Procedure RearrageSparse;
Procedure LU_Solve_by_OpCode;

Implementation

Uses  Declare, LU_Unit;

type  SetOfByte  = Set of Byte;
```

```
    SetArrType = Array[1..1] of SetOfByte;
    SetArrP    = ^SetArrType;

var Row, Col : RCArrP;
  { Row, Col = row and col of the values in ValueP }

   NextRow, NextCol, FirstRow, FirstCol, Diagonal : IndexArrP;
  { NextRow, NextCol = Point to the Next row and next col from the values in ValueP }
  { FirstRow, FirstCol = Point to the first row and the first col in Matrix }
  { Diagonal = Point to the Diagonal Value of Matrix }

   n : IndexType;
  { n = Number of non-zero values }

   RCnt,CCnt  : IndexArrP;
  { RCnt,CCnt  = Count the number of non-zero elements in each rows and columns }

   SPos  :  SetArrP;
  { SPos = Pointer to Array of Set which store where are the non-zero }

   OpCode_LU_Factor : IndexArrP;
  { OpCode_LU_Factor = OpCode for LU_Factor }
  {    ( divisions by pivots )    0, a, b  -> ValueP^[a] := ValueP^[a] / ValueP^[b]; }
  {    ( modifications )          a, b, c  -> ValueP^[a] := ValueP^[a] - ValueP^[b]*ValueP^[c]; }
  { OpCode_LU_Solve  = OpCode for LU_Solve }
  {    ( divisions by pivots )    0, a, b  -> BP^.R[a] := BP^.R[a] / ValueP^[b]; }
  {    ( modifications )          a, b, c  -> BP^.R[a] := BP^.R[a] - ValueP^[b]*BP^.R[c]; }
   OpCode_LU_Factor_Size, OpCode_LU_Factor_Length : IndexType;
  { OpCode_Size   = Size of OpCode_ Array }
  { OpCode_Length = Length of Opcode  ( Length must less then Size ) }
  SparseExtentSize,
  OpCodeExtentSize : Integer; { Size of addition memory for extent Opcode }
  ValuePSeg, ValuePOff,  { Segment and offset of Value }
  BPSeg, BPOff : Word;   { Segment and offset of B }
  ValueP_Shift : Boolean; { Is Offset of Value 0? }
  TempB : MatrixPtr;
  TempB_Shift : Boolean;

Procedure Reordering; Forward;
Procedure ExtentSparse(NewSize:IndexType); Forward;
Procedure Sparse_LU_Factorize; Forward;
Procedure Sparse_LU_Solve; Forward;
Procedure Add_OpCode_LU_Factor( a,b,c:IndexType ); Forward;
Procedure Add_OpCode_LU_Solve( a,b,c:IndexType ); Forward;

Procedure CreateSparse;
var p : Pointer;
begin
 { Sparse Part }
  GetMem(ValueP, SparseSize*SizeOf(Float));
  { Set offset of Value to 8 }
  If Ofs(ValueP^[1]) = 0 then begin
    P := Ptr( Seg(ValueP^[1]), 8);
    ValueP := P;
    Dec(SparseSize);
    ValueP_Shift := True;
  end Else begin
    ValueP_Shift := False;
  end;

  GetMem(Row, (SparseSize+1)*SizeOf(IndexType));
  GetMem(Col, (SparseSize+1)*SizeOf(IndexType));
  GetMem(NextRow, SparseSize*SizeOf(IndexType));
  GetMem(NextCol, SparseSize*SizeOf(IndexType));
  GetMem(FirstRow, Dimension*SizeOf(IndexType));
  GetMem(FirstCol, Dimension*SizeOf(IndexType));
  GetMem(Diagonal, Dimension*SizeOf(IndexType));
  GetMem(RCnt, Dimension*SizeOf(IndexType));
  GetMem(CCnt, Dimension*SizeOf(IndexType));
  GetMem(SPos, Dimension*SizeOf(SetOfByte));
  GetMem(NewR, Dimension*SizeOf(IndexType));
  GetMem(NewC, Dimension*SizeOf(IndexType));
 { Opcode Part }
  UseOpcode := False;
  GetMem(OpCode_LU_Factor, OpCode_LU_Factor_Size*SizePerOpCode*SizeOf(IndexType));
```

```
  GetMem(OpCode_LU_Solve,  OpCode_LU_Solve_Size*SizePerOpCode*SizeOf(IndexType));

  GetMem(TempB, (NumVar1+1)*Sizeof(Float) );
 { Set offset of TempB^[0] to 8 }
  If Ofs(TempB^.R[0]) = 0 then begin
    P := Ptr( Seg(TempB^.R[0]), 8);
    TempB := P;
    TempB_Shift := True;
  end Else begin
    TempB_Shift := False;
  end;
end; { ------------------ CreateSparse --------------- }

Procedure Init_Sparse(d:IndexType);
begin
  ValueP := Nil;
  Dimension  := d;
  Case d of
    1..6 : SparseExtentSize := 2;
  Else
    SparseExtentSize := d*d div 15;
  end;
  SparseSize := SparseExtentSize;
  OpCodeExtentSize := 3*d*d;
  OpCode_LU_Factor_Size := OpCodeExtentSize;
  OpCode_LU_Solve_Size  := OpCodeExtentSize;
  CreateSparse;
  Clear_Sparse;
end; { ---------------- Init_Sparse ----------------- }

Procedure Clear_Sparse;
begin
  n  := 0;
  OpCode_LU_Factor_Length := 0;
  OpCode_LU_Solve_Length  := 0;
  FillChar( FirstRow^, Dimension*SizeOf(IndexType), 0 );
  FillChar( FirstCol^, Dimension*SizeOf(IndexType), 0 );
  FillChar( RCnt^, Dimension*SizeOf(IndexType), $FF );  { -1 }
  FillChar( CCnt^, Dimension*SizeOf(IndexType), $FF );
  FillChar( SPos^, Dimension*SizeOf(SetOfByte), 0 );
  FillChar( NewR^, Dimension*SizeOf(IndexType), 0 );
  FillChar( Diagonal^, Dimension*SizeOf(IndexType), 0 );
end;

Procedure Destroy_Sparse;
var P : Pointer;
begin
  If Diagonal <> Nil then begin
   { Sparse Part }
    If ValueP_Shift then begin
      P := Ptr( Seg(ValueP^[1]), 0);
      ValueP := P;
      FreeMem(ValueP, (SparseSize+1)*SizeOf(Float));
    end Else
      FreeMem(ValueP, SparseSize*SizeOf(Float));
    FreeMem(Row, (SparseSize+1)*SizeOf(IndexType));
    FreeMem(Col, (SparseSize+1)*SizeOf(IndexType));
    FreeMem(NextRow, SparseSize*SizeOf(IndexType));
    FreeMem(NextCol, SparseSize*SizeOf(IndexType));
    FreeMem(FirstRow, Dimension*SizeOf(IndexType));
    FreeMem(FirstCol, Dimension*SizeOf(IndexType));
    FreeMem(Diagonal, Dimension*SizeOf(IndexType));
    FreeMem(RCnt, Dimension*SizeOf(IndexType));
    FreeMem(CCnt, Dimension*SizeOf(IndexType));
    FreeMem(SPos, Dimension*SizeOf(SetOfByte));
    FreeMem(NewR, Dimension*SizeOf(IndexType));
    FreeMem(NewC, Dimension*SizeOf(IndexType));
    Diagonal := Nil;
   { OpCode Part }
    FreeMem(OpCode_LU_Factor, OpCode_LU_Factor_Size*SizePerOpCode*SizeOf(IndexType));
    FreeMem(OpCode_LU_Solve,  OpCode_LU_Solve_Size*SizePerOpCode*SizeOf(IndexType));

    If TempB_Shift then begin
      P := Ptr( Seg(TempB^.R[0]), 0);
      TempB := P;
```

```
  end;
  FreeMem(TempB, (NumVar1+1)*Sizeof(Float) );
 end;
end; { ----------------- Destroy_Sparse ---------------- }

Procedure Input_Data(V:Float; R,C:IndexType);
var i : IndexType;
begin
 If (R=0) or (C=0) {or
   (Abs(V) < NearZero)} then Exit;

 If c in SPos^[r] then begin
  If V = 0 then Exit;
  { Found the old value }
  If RCnt^[r] > CCnt^[c] then begin
   { search in column c }
   Row^[0] := r;     { Residue method }
   i := FirstRow^[c];
   While Row^[i] <> r do
    i := NextRow^[i];
  end Else begin
   { search in row r }
   Col^[0] := c;
   i := FirstCol^[r];
   While Col^[i] <> c do
    i := NextCol^[i];
  end;
  ValueP^[i] := ValueP^[i] + V;
 end Else begin
  If n = SparseSize then { No Space in Sparse Matrix }
   ExtentSparse(SparseSize + SparseExtentSize);
  Inc(n);
  ValueP^[n] := V;
{   NZT^[n] := NonZeroType;}
  Row^[n] := R;
  Col^[n] := C;
  Inc( RCnt^[r] );
  Inc( CCnt^[c] );
  NextCol^[n] := FirstCol^[r];
  FirstCol^[r] := n;
  NextRow^[n] := FirstRow^[c];
  FirstRow^[c] := n;
  SPos^[r] := SPos^[r] + [c];
 end;
end; { -------------------- Input_Data -------------------- }

Function Read_Data(R,C:IndexType) : Float;
var i : IndexType;
begin
 If c in SPos^[r] then begin
  { Found }
  If RCnt^[r] > CCnt^[c] then begin
   { search in column c }
   Row^[0] := r;     { Residue method }
   i := FirstRow^[c];
   While Row^[i] <> r do
    i := NextRow^[i];
  end Else begin
   { search in row r }
   Col^[0] := c;
   i := FirstCol^[r];
   While Col^[i] <> c do
    i := NextCol^[i];
  end;
  Read_Data := ValueP^[i];
 end Else Read_Data := 0;
end; { --------------------- Read_Data -------------------- }

Procedure ExtentSparse( NewSize:IndexType );
var VP : ValueArrP;
  R,C : RCArrP;
  NR,NC : IndexArrP;
{   NZ : NZTypeArrP;}
  p : Pointer;
  VP_Shift : Boolean;
```

```
begin
 GetMem(VP, NewSize*SizeOf(Float));
 { Set offset of V to 8 }
 If Ofs(VP^[1]) = 0 then begin
  P := Ptr( Seg(VP^[1]), 8);
  VP := P;
  Dec(NewSize);
  VP_Shift := True;
 end Else begin
  VP_Shift := False;
 end;
 Move( ValueP^, VP^, SparseSize*SizeOf(Float) );
 If ValueP_Shift then begin
  P := Ptr( Seg(ValueP^[1]), 0);
  ValueP := P;
  FreeMem(ValueP, (SparseSize+1)*SizeOf(Float));
 end Else
  FreeMem(ValueP, SparseSize*SizeOf(Float));
 ValueP := VP;
 ValueP_Shift := VP_Shift;

 GetMem(R, (NewSize+1)*SizeOf(IndexType));
 Move( Row^, R^, (SparseSize+1)*SizeOf(IndexType) );
 FreeMem(Row, (SparseSize+1)*SizeOf(IndexType));
 Row := R;

 GetMem(C, (NewSize+1)*SizeOf(IndexType));
 Move( Col^, C^, (SparseSize+1)*SizeOf(IndexType) );
 FreeMem(Col, (SparseSize+1)*SizeOf(IndexType));
 Col := C;

 GetMem(NR, NewSize*SizeOf(IndexType));
 Move( NextRow^, NR^, SparseSize*SizeOf(IndexType) );
 FreeMem(NextRow, SparseSize*SizeOf(IndexType));
 NextRow := NR;

 GetMem(NC, NewSize*SizeOf(IndexType));
 Move( NextCol^, NC^, SparseSize*SizeOf(IndexType) );
 FreeMem(NextCol, SparseSize*SizeOf(IndexType));
 NextCol := NC;

 SparseSize := NewSize;
end; { ---------------- Extent_sparse -------------- }

Function Finish_Input_Data : Boolean;
 { Return  True  = Solve completely }
 {         False = Pivot = 0 while solve by opcode }
begin
 If Not UseOpCode then begin
  Opcode_LU_Solve_Length := 0;
  Row^[0] := Dimension+1;
  Col^[0] := Dimension+1;

  { Solve Matrix & Build OpCode }
  Reordering;
  Singular := False;
  FloatOverFlow := False;

  Sparse_LU_Factorize;
  If Not(Singular or FloatOverFlow) then
   Sparse_LU_Solve;

  UseOpCode := True;
 end Else begin
  LU_Solve_by_Opcode;
 end;
 Finish_Input_Data := UseOpcode;
end; { ------------------ Finish_Input_Data ----------------- }

Procedure Reordering;
var NewFill : IndexArrP;
  NewFCnt : IndexType;

 Procedure Markowitz;
 var iMin, a,
```

```
  i, j, r  : IndexType;
  MinV, V  : Integer;

begin
 For a := 1 to Dimension-1 do begin
  MinV := MaxInt;
  { Find Min Value }
  For r := 1 to Dimension do
   If NewR^[r] = 0 then begin
    i := FirstCol^[r];
    While i <> 0 do begin
     If NewC^[ Col^[i] ] = 0 then begin
      V := RCnt^[r] * CCnt^[ Col^[i] ];
      If (V<MinV) or
        (V = MinV)and(RCnt^[r]<RCnt^[Row^[iMin]]) or
        (V = MinV)and(RCnt^[r]=RCnt^[Row^[iMin]])and(Abs(ValueP^[i])>Abs(ValueP^[iMin]))
      then If Abs(ValueP^[i])>NearZero then begin
          iMin := i;
          MinV := V;
         end;
     end;
     i := NextCol^[i];
    end;
   end;
  { end of for r }

  NewR^[ Row^[iMin] ] := a;
  NewC^[ Col^[iMin] ] := a;

  { Store the column of non-zero in Row^[iMin] into NewFill }
  NewFCnt := 0;
  i := FirstCol^[ Row^[iMin] ];
  While i <> 0 do begin
   If NewC^[ Col^[i] ] = 0 then begin
    Inc( NewFCnt );
    NewFill^[ NewFCnt ] := Col^[i];
    Dec( CCnt^[ Col^[i] ] );
   end;
   i := NextCol^[i];
  end;

  { Check new fill-ins }
  i := FirstRow^[ Col^[iMin] ];
  While i <> 0 do begin
   If NewR^[ Row^[i] ] = 0 then begin
    For j := 1 to NewFCnt do begin
     r := n;   { use r as temp var }
     Input_Data( 0, Row^[i], NewFill^[j] );
     If r < n then { If there is a new fill-ins }
{}     Add_OpCode_LU_Factor( 8,n,0 );   { ValueP^[n] := 0 }
    end;
    Dec( RCnt^[ Row^[i] ] );
   end;
   i := NextRow^[i];
  end;
 end; { for a }
end;

Procedure ChangeRowCol;
var i,r : IndexType;
begin
 i := 1;
 While NewR^[i] <> 0 do Inc(i);
 NewR^[i] := Dimension;

 i := 1;
 While NewC^[i] <> 0 do Inc(i);
 NewC^[i] := Dimension;

 For r := 1 to Dimension do begin
  i := FirstCol^[r];
  While i <> 0 do begin
   Row^[i] := NewR^[r];
   Col^[i] := NewC^[ Col^[i] ];
   i := NextCol^[i];
```

```
      end;
     { Use FirstRow as the Temp Array }
      FirstRow^[ NewR^[r] ] := FirstCol^[r];
    end;

    For r := 1 to Dimension do
      FirstCol^[r] := FirstRow^[r];

  end;

{ Main of Reordering }
begin
  FillChar( NewC^, Dimension*SizeOf(IndexType), 0 );
  GetMem( NewFill, Dimension*SizeOf(IndexType) );
  ValuePOff := Ofs( ValueP^[1] );
  Markowitz;
  ChangeRowCol;
  RearrageSparse;
  FreeMem( NewFill, Dimension*SizeOf(IndexType) );
end; {-------------------- Reordering --------------------}

Procedure RearrageSparse;
var A : IndexArrP;
    r : IndexType;

 Procedure LinkCol;
 var c, i : Integer;
 begin
   FillChar( A^, Dimension*SizeOf(IndexType), 0 );
   i := FirstCol^[r];
   While i <> 0 do begin
     c := Col^[i];
{}     Inc( RCnt^[r] );
{}     Inc( CCnt^[c] );
{}     SPos^[r] := SPos^[r] + [c];
     A^[c] := i;
     i := NextCol^[i];
   end;
   { Link Column in Row "R" }
   FirstCol^[r] := 0;
   For c := Dimension DownTo 1 do
     If A^[c] <> 0 then begin
       NextCol^[ A^[c] ] := FirstCol^[r];
       FirstCol^[r] := A^[c];
       NextRow^[ A^[c] ] := FirstRow^[c];
       FirstRow^[c] := A^[c];
     end;
   Diagonal^[r] := A^[r];
 end;

{ Main of RearrageSparse }
begin
  GetMem( A , Dimension*SizeOf(IndexType) );
  FillChar( FirstRow^, Dimension*SizeOf(IndexType), 0 );
{}  FillChar( RCnt^, Dimension*SizeOf(IndexType), $FF );  { -1 }
{}  FillChar( CCnt^, Dimension*SizeOf(IndexType), $FF );
{}  FillChar( SPos^, Dimension*SizeOf(SetOfByte), 0 );
  For r := Dimension Downto 1 do
    LinkCol;
  FreeMem( A, Dimension*SizeOf(IndexType) );
end; {----------------- RearrageSparse ----------------- }

{----------------------- LU_Factorize Part -----------------------}
Procedure Sparse_LU_Factorize;
var FirstMul : Boolean;

 Function Cal_Sigma(i,r,c,a:IndexType) : Double;
 var k,m : IndexType;
     Sigma  : Double;
 begin
   k := FirstRow^[c];  m := FirstCol^[r];
   Sigma := 0;
   FirstMul := True;
   While (Row^[k]<a) and (Col^[m]<a) do begin
     If Row^[k] = Col^[m] then begin
```

```
       Sigma := Sigma + ValueP^[k]*ValueP^[m];
{}       Add_OpCode_LU_Factor( 1,k,m );   { st = k*m }
       If Not FirstMul then
{}         Add_OpCode_LU_Factor( 3,0,0 ); { st = st+st(1) }
       FirstMul := False;

       k := NextRow^[k];
       m := NextCol^[m];
     end Else begin
       While Row^[k] > Col^[m] do
         m := NextCol^[m];
       If Col^[m] >= a then begin
         Cal_Sigma := Sigma;
         Exit;
       end;
       While Row^[k] < Col^[m] do
         k := NextRow^[k];
     end;
   end;
   If Abs(Sigma) > MaxFloat then begin
     FloatOverFlow := True;
     Exit;
   end;
   Cal_Sigma := Sigma;
 end;   { st = Sigma value }

var i,a  : IndexType;
begin
  If (Abs(ValueP^[ Diagonal^[1] ]) < NearZero)
     or (n<Dimension) then begin
    Singular := True;
    Exit;
  end;

  ValuePOff := Ofs( ValueP^[1] );

{}Add_OpCode_LU_Factor( 5,Diagonal^[1],0 ); { st   = D[1] }
{}Add_OpCode_LU_Factor( 6,0,0 );            { st   = 1/st }
{}Add_OpCode_LU_Factor( 7,Diagonal^[1],0 );  { D[1] = st }

 { Cal U in 1st Row }
  i := NextCol^[ Diagonal^[1] ];
  While i <> 0 do begin
    ValueP^[i] := ValueP^[i]/ValueP^[ Diagonal^[1] ];
{}    Add_OpCode_LU_Factor( 1,i,Diagonal^[1] ); { st = i*D[1] }
{}    Add_OpCode_LU_Factor( 7,i,0 );            { i  = st }
    i := NextCol^[i];
  end;
 { and Cal in other row }
  For a := 2 to Dimension do begin
    i := Diagonal^[a];
   { Cal L }
    While i <> 0 do begin
      ValueP^[i] := ValueP^[i] - Cal_Sigma(i, Row^[i], a, a );  { st = Sigma of Multipulations }
      If Not FirstMul then begin { There is some multiplication }
{}        Add_OpCode_LU_Factor( 4,i,0 );   { st = i - st }
        If i = Diagonal^[a] then
{}          Add_OpCode_LU_Factor( 6,0,0 ); { st = 1/st }
{}        Add_OpCode_LU_Factor( 7,i,0 );   { i  = st }
      end Else
        If i = Diagonal^[a] then begin
{}          Add_OpCode_LU_Factor( 5,i,0 ); { st = i }
{}          Add_OpCode_LU_Factor( 6,0,0 ); { st = 1/st }
{}          Add_OpCode_LU_Factor( 7,i,0 ); { i  = st }
        end;
      i := NextRow^[i];
    end;
    If (Diagonal^[a] = 0) or
       (Abs(ValueP^[ Diagonal^[a] ]) < NearZero) then begin
      Singular := True;
      Exit;
    end;
    If FloatOverFlow then Exit;

   { Cal U }
```

```
  i := NextCol^[ Diagonal^[a] ];
  While i <> 0 do begin
   ValueP^[i] := (ValueP^[i] - Cal_Sigma(i, a, Col^[i], a ))/ValueP^[ Diagonal^[a] ];
   If Not FirstMul then begin { There is some multiplication }
{}     Add_OpCode_LU_Factor( 4,i,0 );           { st = i-st }
{}     Add_OpCode_LU_Factor( 2,i,Diagonal^[a] ); { i  = D[a]*st }
   end Else begin
{}     Add_OpCode_LU_Factor( 1,i,Diagonal^[a] ); { st = i*D[a] }
{}     Add_OpCode_LU_Factor( 7,i,0 );           { i  = st }
   end;
   i := NextCol^[i];
  end;
  If FloatOverFlow then Exit;
 end;
end;{------------------- LU_Factorize Part -----------------------}

{----------------------- LU_Solve Part ------------------------}
Procedure Sparse_LU_Solve;
var OldR : IndexArrP;
  FirstMul : Boolean;

 Procedure Forward_substistution;
 var r,m : IndexType;
   V   : Double;
 begin
  For r := 1 to Dimension do begin
    m := FirstCol^[r];
    V := 0;
    FirstMul := True;
    While Col^[m] < r do begin
     V := V + BP^.R[ Col^[m] ]*ValueP^[m];
{}      Add_OpCode_LU_Solve( 1, OldR^[Col^[m]], m );  { st = B[C[m]]*m }
     If Not FirstMul then
{}        Add_OpCode_LU_Solve( 3, 0, 0 );            { st = st+st(1) }
     FirstMul := False;
     m := NextCol^[m];
    end;

    If Not FirstMul then begin { There is some multiplication }
     V := BP^.R[r] - V;
{}      Add_OpCode_LU_Solve( 4, OldR^[r], 0 );       { st = B[r]-st }
      BP^.R[r] := V/ValueP^[ Diagonal^[r] ];
{}      Add_OpCode_LU_Solve( 2, OldR^[r], Diagonal^[r] ); { B[r] = D[r]*st }
    end Else begin
     BP^.R[r] := BP^.R[r]/ValueP^[ Diagonal^[r] ];
{}      Add_OpCode_LU_Solve( 1, OldR^[r], Diagonal^[r] ); { st = B[r]*D[r] }
{}      Add_OpCode_LU_Solve( 7, OldR^[r], 0 );           { B[r]  = st }
    end;
   end;
 end;

 Procedure Back_substistution;
 var r,m : IndexType;
   V : Double;
 begin
  For r := Dimension downto 1 do begin
   { If Diagonal^[r] <> 0 then ;}
   V := 0;
   FirstMul := True;
   m := NextCol^[ Diagonal^[r] ];
   While Col^[m] < Dimension+1 do begin
    V := V + BP^.R[ Col^[m] ]*ValueP^[m];
{}    Add_OpCode_LU_Solve( 1, OldR^[Col^[m]], m ); { st = B[C[m]]*m }
    If Not FirstMul then
{}      Add_OpCode_LU_Solve( 3, 0, 0 );           { st = st+st(1) }
    FirstMul := False;
    m := NextCol^[m];
   end;

   If Not FirstMul then begin
    V := BP^.R[r] - V;
{}    Add_OpCode_LU_Solve( 4, OldR^[r], 0 ); { st = B[r] - st }

    BP^.R[r] := V;
{}    Add_OpCode_LU_Solve( 7, OldR^[r], 0 ); { B[r]  = st }
```

```
   end;
  end;

{}  Add_OpCode_LU_Solve( 9, 0, 0 );  { RETF }
 end;

var TempB,TB : MatrixPtr ;
  SizeofNum:byte;
  NumVar1Size:Integer;
  r,c : IndexType;
  P : Pointer;
  BP_Shift : Boolean;

 procedure ClearMem;
 begin
  FreeMem( OldR, Dimension*SizeOf(IndexType) );
  If BP_Shift then begin
   P := Ptr( Seg(BP^.R[0]), 0);
   BP := P;
  end;
  FreeMem( BP, NumVar1Size );
 end;

{ Main of Sparse_LU_Solve }
begin
 Row^[0] := Dimension+1;
 Col^[0] := Dimension+1;
 GetMem( OldR, Dimension*SizeOf(IndexType) );
 For r := 1 to Dimension do
  OldR^[ NewR^[r] ] := r;

 { Swap B ordering by NewR }
 if OpMode=ACsim then SizeofNum:=SizeOf(Complex)
 else                SizeofNum:=SizeOf(Float);
 NumVar1Size:=(NumVar1+1)*SizeofNum;
 GetMem(TempB,NumVar1Size);
 { Set offset of TempB^[0] to 8 }
 If Ofs(TempB^.R[0]) = 0 then begin
  P := Ptr( Seg(TempB^.R[0]), 8);
  TempB := P;
  BP_Shift := True;
 end Else begin
  BP_Shift := False;
 end;
 For r := 1 to Dimension do
  TempB^.R[ NewR^[r] ] := BP^.R[r];
 TB := BP;
 BP := TempB;
 TempB := TB;

 ValuePOff := Ofs( ValueP^[1] );
 BPOff := Ofs( BP^.R[0] );

 forward_substistution;
 If FloatOverFlow then begin
  ClearMem;
  Exit;
 end;
 back_substistution;
 If FloatOverFlow then begin
  ClearMem;
  Exit;
 end;

 { Swap B ordering by NewC }
 For c := 1 to Dimension do
  TempB^.R[ c ] := BP^.R[ NewC^[c] ];

 { Change value in NewC by use NewR like TempArray }
 For c := 1 to Dimension do
  NewR^[ c ] := OldR^[ NewC^[c] ];

 ClearMem;
 BP := TempB;
 BP^.R[0] := 0.0;
```

```
  OldR := NewC;
  NewC := NewR;
  NewR := OldR;

 (* 16/11/42 *)
 { Store the inverse value of diagonal A in the head of ValueP }
 { For Matrix cache only }
  For c := 1 to Dimension do
    ValueP^[ Diagonal^[c] ] := 1/ValueP^[ Diagonal^[c] ];
end;{-------------------- Sparse_LU_Solve Part ---------------------}

{--------------------------------- OpCode Part -----------------------------}
Procedure ExtentOpCode( Var OpCode:IndexArrP;
                        Var OpCodeSize:IndexType;
                        NewSize:IndexType );
var TempOpCode : IndexArrP;
begin
  GetMem(TempOpCode, NewSize*SizePerOpCode*SizeOf(IndexType));
  Move( OpCode^, TempOpCode^, OpCodeSize*SizePerOpCode*SizeOf(IndexType) );
  FreeMem(OpCode, OpCodeSize*SizePerOpCode*SizeOf(IndexType));
  OpCode := TempOpCode;
  OpCodeSize := NewSize;
end;

Procedure Add_OpCode_LU_Factor( a,b,c:IndexType );
begin
(***** Machine Code *****)
  b := (b-1)*SizeOf(Float) + ValuePOff;
  c := (c-1)*SizeOf(Float) + ValuePOff;
  Case a of
   1 : begin { st := ValueP^[b]*ValueP^[c] }
        If OpCode_LU_Solve_Length+4  > OpCode_LU_Solve_Size then
          ExtentOpCode( OpCode_LU_Solve, OpCode_LU_Solve_Size, OpCode_LU_Solve_Size+OpCodeExtentSize );
       { FLD    QWORD ValueP^[b] }
        OpCode_LU_Solve^[OpCode_LU_Solve_Length+1] := $06DD;
        OpCode_LU_Solve^[OpCode_LU_Solve_Length+2] := b;
       { FMUL   QWORD ValueP^[c] }
        OpCode_LU_Solve^[OpCode_LU_Solve_Length+3] := $0EDC;
        OpCode_LU_Solve^[OpCode_LU_Solve_Length+4] := c;

        Inc(OpCode_LU_Solve_Length,4);
      end;
   2 : begin { ValueP^[b] := st*ValueP^[c] }
        If OpCode_LU_Solve_Length+4 > OpCode_LU_Solve_Size then
          ExtentOpCode( OpCode_LU_Solve, OpCode_LU_Solve_Size, OpCode_LU_Solve_Size+OpCodeExtentSize );
       { FMUL   QWORD ValueP^[c] }
        OpCode_LU_Solve^[OpCode_LU_Solve_Length+1] := $0EDC;
        OpCode_LU_Solve^[OpCode_LU_Solve_Length+2] := c;
       { FSTP   QWORD ValueP^[b] }
        OpCode_LU_Solve^[OpCode_LU_Solve_Length+3] := $1EDD;
        OpCode_LU_Solve^[OpCode_LU_Solve_Length+4] := b;

        Inc(OpCode_LU_Solve_Length,4);
      end;
   3 : begin { st := st+st(1) }
        If OpCode_LU_Solve_Length+1 > OpCode_LU_Solve_Size then
          ExtentOpCode( OpCode_LU_Solve, OpCode_LU_Solve_Size, OpCode_LU_Solve_Size+OpCodeExtentSize );
       { FADDP  ST(1) }
        OpCode_LU_Solve^[OpCode_LU_Solve_Length+1] := $C1DE;

        Inc(OpCode_LU_Solve_Length,1);
      end;
   4 : begin { st := ValueP^[b] - st }
        If OpCode_LU_Solve_Length+2 > OpCode_LU_Solve_Size then
          ExtentOpCode( OpCode_LU_Solve, OpCode_LU_Solve_Size, OpCode_LU_Solve_Size+OpCodeExtentSize );
       { FSUBR  QWORD ValueP^[b] }
        OpCode_LU_Solve^[OpCode_LU_Solve_Length+1] := $2EDC;
        OpCode_LU_Solve^[OpCode_LU_Solve_Length+2] := b;

        Inc(OpCode_LU_Solve_Length,2);
      end;
   5 : begin { ST = ValueP^[b] }
        If OpCode_LU_Solve_Length+2 > OpCode_LU_Solve_Size then
          ExtentOpCode( OpCode_LU_Solve, OpCode_LU_Solve_Size, OpCode_LU_Solve_Size+OpCodeExtentSize );
```

```
      { FLD    QWORD ValueP^[b] }
       OpCode_LU_Solve^[OpCode_LU_Solve_Length+1] := $06DD;
       OpCode_LU_Solve^[OpCode_LU_Solve_Length+2] := b;

       Inc(OpCode_LU_Solve_Length,2);
     end;
  6 : begin { st := 1 / st }
       If OpCode_LU_Solve_Length+7 > OpCode_LU_Solve_Size then
         ExtentOpCode( OpCode_LU_Solve, OpCode_LU_Solve_Size, OpCode_LU_Solve_Size+OpCodeExtentSize );

      { FTST }
       OpCode_LU_Solve^[OpCode_LU_Solve_Length+1] := $E4D9;
      { FNSTSW AX }
       OpCode_LU_Solve^[OpCode_LU_Solve_Length+2] := $E0DF;
      { AND AH,40h }
       OpCode_LU_Solve^[OpCode_LU_Solve_Length+3] := $E480;
      { If st = 0 then }
       OpCode_LU_Solve^[OpCode_LU_Solve_Length+4] := $7440;
      { RETF  Else }
       OpCode_LU_Solve^[OpCode_LU_Solve_Length+5] := $CB01;
      { FLD1 }
       OpCode_LU_Solve^[OpCode_LU_Solve_Length+6] := $E8D9;
      { FDIVRP ST(1) }
       OpCode_LU_Solve^[OpCode_LU_Solve_Length+7] := $F1DE;

       Inc(OpCode_LU_Solve_Length,7);
      end;
  7 : begin { ValueP^[b] := st }
       If OpCode_LU_Solve_Length+2 > OpCode_LU_Solve_Size then
         ExtentOpCode( OpCode_LU_Solve, OpCode_LU_Solve_Size, OpCode_LU_Solve_Size+OpCodeExtentSize );
      { FSTP  QWORD ValueP^[b] }
       OpCode_LU_Solve^[OpCode_LU_Solve_Length+1] := $1EDD;
       OpCode_LU_Solve^[OpCode_LU_Solve_Length+2] := b;

       Inc(OpCode_LU_Solve_Length,2);
     end;
  8 : begin { ValueP^[b] := 0 }
       If OpCode_LU_Solve_Length+3 > OpCode_LU_Solve_Size then
         ExtentOpCode( OpCode_LU_Solve, OpCode_LU_Solve_Size, OpCode_LU_Solve_Size+OpCodeExtentSize );
      { FLDZ }
       OpCode_LU_Solve^[OpCode_LU_Solve_Length+1] := $EED9;
      { FSTP QWORD ValueP^[b] }
       OpCode_LU_Solve^[OpCode_LU_Solve_Length+2] := $1EDD;
       OpCode_LU_Solve^[OpCode_LU_Solve_Length+3] := b;

       Inc(OpCode_LU_Solve_Length,3);
     end;
  9 : begin { Return Far }
       If OpCode_LU_Solve_Length+2 > OpCode_LU_Solve_Size then
         ExtentOpCode( OpCode_LU_Solve, OpCode_LU_Solve_Size, OpCode_LU_Solve_Size+OpCodeExtentSize );
      { MOV AX,$1234 }  { Mark if complete calculation }
       OpCode_LU_Solve^[OpCode_LU_Solve_Length+1] := $34B8;
      { RETF }
       OpCode_LU_Solve^[OpCode_LU_Solve_Length+2] := $CB12;
       Inc(OpCode_LU_Solve_Length,2);
     end;
 end; { Case a }

end;{-------------------Add_OpCode_LU_Factor-------------------------}

Procedure Add_OpCode_LU_Solve( a,b,c:IndexType );
var FloatSize : Word;
begin
(***** Machine Code *****)
 { DS store segment of ValueP^ }
 { ES store segment of BP^ }
 FloatSize := SizeOf(Float);
 b := b*FloatSize + BPOff;
 c := (c-1)*FloatSize + ValuePOff;
 Case a of
  1 : begin { st := BP^[b]*ValueP^[c] }
       If OpCode_LU_Solve_Length+5  > OpCode_LU_Solve_Size then
         ExtentOpCode( OpCode_LU_Solve, OpCode_LU_Solve_Size, OpCode_LU_Solve_Size+OpCodeExtentSize );
      { ES: }
```

```
    OpCode_LU_Solve^[OpCode_LU_Solve_Length+1] := $2690;
    { FLD    QWORD BP^[b] }
    OpCode_LU_Solve^[OpCode_LU_Solve_Length+2] := $06DD;
    OpCode_LU_Solve^[OpCode_LU_Solve_Length+3] := b;
    { FMUL   QWORD ValueP^[c] }
    OpCode_LU_Solve^[OpCode_LU_Solve_Length+4] := $0EDC;
    OpCode_LU_Solve^[OpCode_LU_Solve_Length+5] := c;

    Inc(OpCode_LU_Solve_Length,5);
   end;
 2 : begin { BP^[b] := st*ValueP^[c] }
    If OpCode_LU_Solve_Length+5 > OpCode_LU_Solve_Size then
     ExtentOpCode( OpCode_LU_Solve, OpCode_LU_Solve_Size, OpCode_LU_Solve_Size+OpCodeExtentSize );
    { FMUL   QWORD ValueP^[c] }
    OpCode_LU_Solve^[OpCode_LU_Solve_Length+1] := $0EDC;
    OpCode_LU_Solve^[OpCode_LU_Solve_Length+2] := c;
    { ES: }
    OpCode_LU_Solve^[OpCode_LU_Solve_Length+3] := $2690;
    { FSTP   QWORD BP^[b] }
    OpCode_LU_Solve^[OpCode_LU_Solve_Length+4] := $1EDD;
    OpCode_LU_Solve^[OpCode_LU_Solve_Length+5] := b;

    Inc(OpCode_LU_Solve_Length,5);
   end;
 3 : begin { st := st+st(1) }
    If OpCode_LU_Solve_Length+1 > OpCode_LU_Solve_Size then
     ExtentOpCode( OpCode_LU_Solve, OpCode_LU_Solve_Size, OpCode_LU_Solve_Size+OpCodeExtentSize );
    { FADDP  ST(1) }
    OpCode_LU_Solve^[OpCode_LU_Solve_Length+1] := $C1DE;

    Inc(OpCode_LU_Solve_Length,1);
   end;
 4 : begin { st := BP^[b] - st }
    If OpCode_LU_Solve_Length+3 > OpCode_LU_Solve_Size then
     ExtentOpCode( OpCode_LU_Solve, OpCode_LU_Solve_Size, OpCode_LU_Solve_Size+OpCodeExtentSize );
    { ES: }
    OpCode_LU_Solve^[OpCode_LU_Solve_Length+1] := $2690;
    { FSUBR  QWORD BP^[b] }
    OpCode_LU_Solve^[OpCode_LU_Solve_Length+2] := $2EDC;
    OpCode_LU_Solve^[OpCode_LU_Solve_Length+3] := b;

    Inc(OpCode_LU_Solve_Length,3);
   end;
7 : begin { BP^[b] := st }
    If OpCode_LU_Solve_Length+3 > OpCode_LU_Solve_Size then
     ExtentOpCode( OpCode_LU_Solve, OpCode_LU_Solve_Size, OpCode_LU_Solve_Size+OpCodeExtentSize );
    { ES: }
    OpCode_LU_Solve^[OpCode_LU_Solve_Length+1] := $2690;
    { FSTP  QWORD BP^[b] }
    OpCode_LU_Solve^[OpCode_LU_Solve_Length+2] := $1EDD;
    OpCode_LU_Solve^[OpCode_LU_Solve_Length+3] := b;

    Inc(OpCode_LU_Solve_Length,3);
   end;
 8 : begin { ValueP^[b] := 0 }
    If OpCode_LU_Solve_Length+3 > OpCode_LU_Solve_Size then
     ExtentOpCode( OpCode_LU_Solve, OpCode_LU_Solve_Size, OpCode_LU_Solve_Size+OpCodeExtentSize );
    { FLDZ }
    OpCode_LU_Solve^[OpCode_LU_Solve_Length+1] := $EED9;
    { FSTP QWORD ValueP^[b] }
    OpCode_LU_Solve^[OpCode_LU_Solve_Length+2] := $1EDD;
    OpCode_LU_Solve^[OpCode_LU_Solve_Length+3] := b;

    Inc(OpCode_LU_Solve_Length,3);
   end;
 9 : begin { Return Far }
    If OpCode_LU_Solve_Length+2 > OpCode_LU_Solve_Size then
     ExtentOpCode( OpCode_LU_Solve, OpCode_LU_Solve_Size, OpCode_LU_Solve_Size+OpCodeExtentSize );
    { MOV AX,$1234 }  { Mark if complete calculation }
    OpCode_LU_Solve^[OpCode_LU_Solve_Length+1] := $34B8;
    { RETF }
    OpCode_LU_Solve^[OpCode_LU_Solve_Length+2] := $CB12;
    Inc(OpCode_LU_Solve_Length,2);
   end;
end; { Case a }
```

```
end;

Procedure LU_Factor_by_OpCode;
begin
end; {---------------------- LU_Factor_by_OpCode ------------------------}

Procedure LU_Solve_by_OpCode;

var SizeofNum:byte;
    i,c : IndexType;
    InstSeg,InstOff : Word;
    P : Pointer;
begin
  InstSeg := Seg( OpCode_LU_Solve^[1] );
  InstOff := Ofs( OpCode_LU_Solve^[1] );
  StartInst[1] := $9A;   { Call Far OpCode_LU_Solve^ }
  StartInst[2] := Lo( InstOff );
  StartInst[3] := Hi( InstOff );
  StartInst[4] := Lo( InstSeg );
  StartInst[5] := Hi( InstSeg );
  StartInst[6] := $CB;   { Return Far }

  Move( BP^, TempB^, NumVar1*SizeOf(Float) );

  ValuePSeg := Seg( ValueP^[1] );
  BPSeg := Seg( TempB^.R[0] );

  Inc( CHit );
  i := 0;

  asm
    MOV   AX,$0000
    PUSH  DS
    PUSH  ES
    MOV   ES,BPSeg
    MOV   DS,ValuePSeg

    call far [StartInst]

    POP   ES
    POP   DS
    MOV   i,AX
  end;

  If i <> $1234 then begin { PIVOT is Zero }
    UseOpcode := False;
    Exit;
  end;

 { Swap B ordering by NewC }
  For c := 1 to Dimension do
    BP^.R[ c ] := TempB^.R[ NewC^[c] ];
  BP^.R[0] := 0.0;
end; {---------------------- LU_Solve_by_OpCode ------------------------}

Begin
  Diagonal := Nil;
End.
```

ประวัติผู้วิจัย

นายนภดล จิตต์จรัส เกิดวันที่ 9 พฤษภาคม พ.ศ.2520 ที่เขตพญาไท กรุงเทพมหานคร สำเร็จการศึกษาปริญญาตรีวิศวกรรมศาสตรบัณฑิต สาขาวิศวกรรมไฟฟ้า คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย ในปีการศึกษา 2540 เข้าศึกษาต่อในหลักสูตรวิศวกรรมศาสตรมหาบัณฑิต สาขาวิศวกรรมไฟฟ้า (แขนงวิชาระบบเชิงเลข) ที่คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย เมื่อปี พ.ศ. 2541 และได้รับทุนการศึกษาจากโครงการศิษย์ก้นกุฏิ ตั้งแต่ภาคการศึกษาปลาย ปีการศึกษา 2541