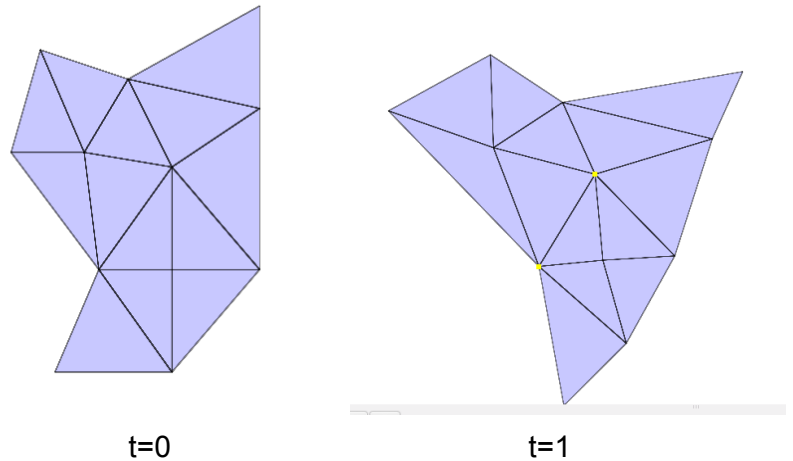


# מימוש של BDMORPH

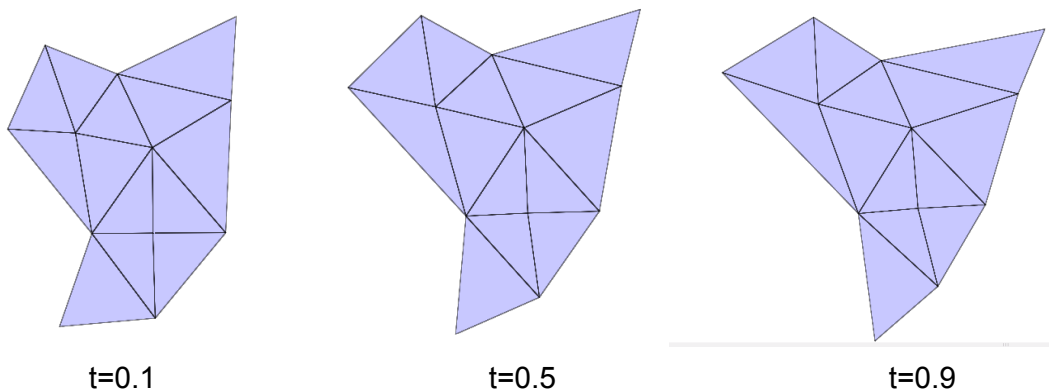
## הקלט של אלגוריתם והפלט שלו

הקלט של אלגוריתם הוא שני mesh אשר בשניהם יש אותו מספר של קודקודים וקשתות. בעצם מש אחד הוא זהה למש שני עד כדי הזזה של נקודת. את מש השני יוצרים על ידי גרירה של נקודת והפעלה של KVF כמו שאנו מכירים.



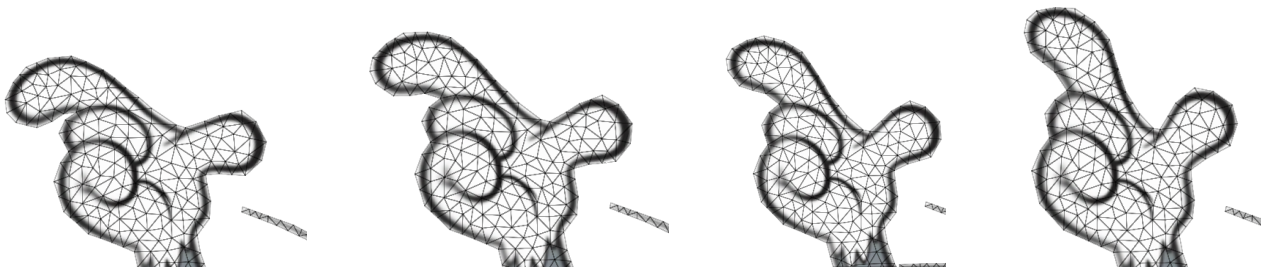
המטרה של אלגוריתם היא למצוא מש ביניים כתלות בפרמטר קלט  $t$  אשר מייצג זמן מנורמל. עבור  $t=0$  נקבל מש ראשון. עבור  $t$  ממש קטן נקבל כמעט מש ראשון. עבור  $t=0.5$  נקבל מש דומה לשני המשים עבור  $t$  קרוב ל 1 נקבל כמעט מש שני ועבור  $t=1$  נקבל בדיוק מש שני.

דוגמאות:



במקרה שלמש יש תמונה כל משולש נמתח בהתאמה וזה יותר תחושה של אנימציה:





כפי שרואים אפילו שנדמה שתמונה משתנה מה שמשתנה זה מיקומים של קדקודים וה texture נמתח מחדש על משולשים הללו.

## הסבר הכללי מאחורי עקרון של אלגוריתם

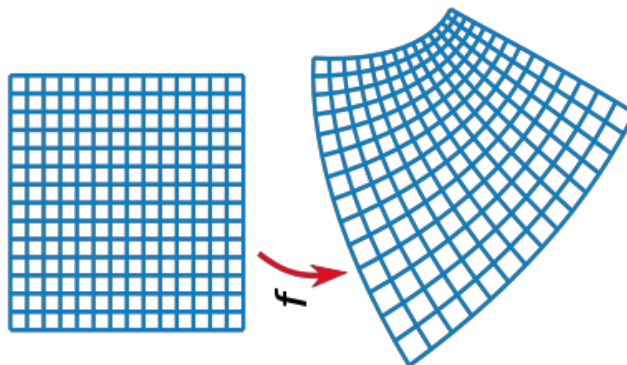
אלגוריתם מבוסס על אינטרפולציה של מטריקה. מטריקה בהקשר שלנו זה אוסף של מרחקים בין קודקודים של המש.

אבל מרחקים הללו לא חייבים להיות מרחקים כמו שאנו מכירים אותם. זה יכולים להיות מספרים כמעט כול שהם כל עוד הם מקיימים דרישות הבאות:

1. מרחק תמיד חיובי, ורק מרחק של נקודה מעצמה הוא 0
2. אי שיוויון של משולש מתקיים (מרחק בין שני נקודות קטן מן סכום מרחקים של שניהם מנקודה שלישית)
3. וסותם פורמלית מרחק לא תלוי בסדר של נקודות.

אחד המדדים של איכות של דיפורמציה של מש וגם בפרט אינטרפולציה שלה זה עיוות זוויתי (conformal distortion). בהקשר שלנו זה בעצם אומר עד כמה הזווית של משולשים במש הביניים שקיבלנו שונות מהזווית של שני המשים. באופן יותר רחב המושג זה מדבר על שימור של זוויות בסביבה של נקודה.

אפשר להגיד שיש שמירה על זווית (או קונפורמיות) אם עבור כל שני עקומים בתמונה ראשונה הזווית החיתוך שלהם נשאר זהה בסביבה של נקודה זאת. דוגמה מוויקיפדיה:



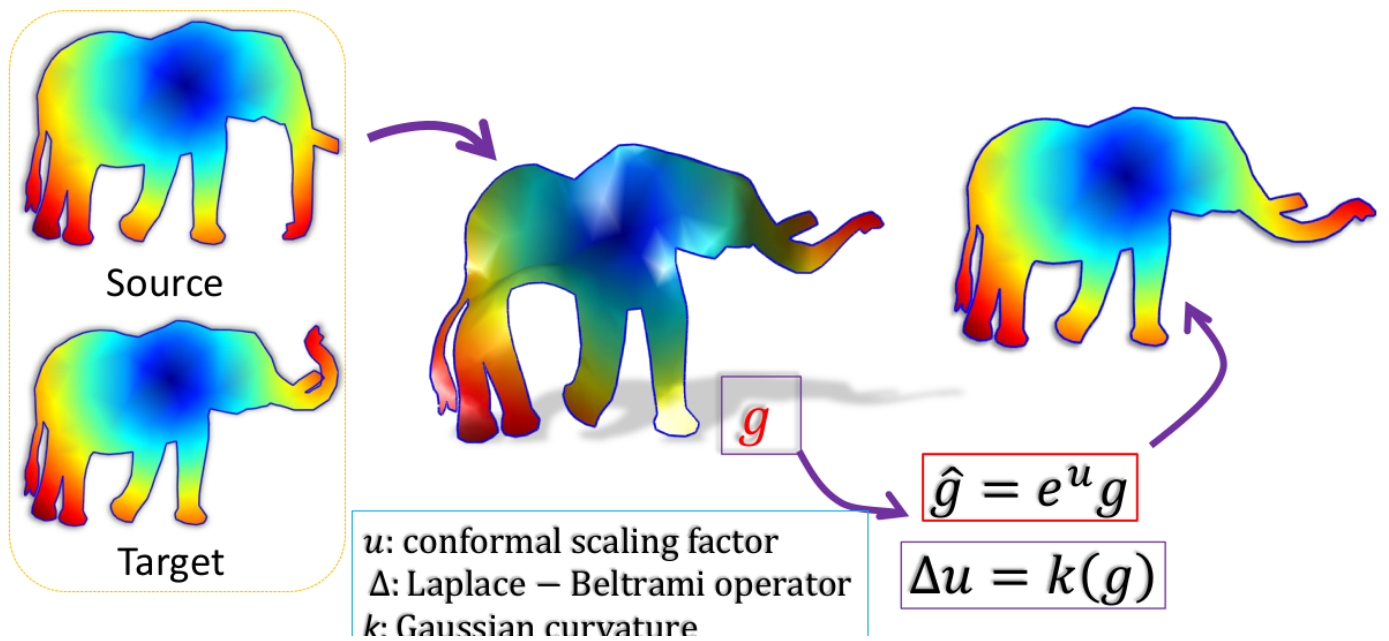
שני משים הללו שונים זה מזה אבל עבור כל עקום הזווית נשמרת כבערך 90 מעלות.

העיקרון של מאמר הוא לחשב את שני מטריקות של שני המשים ואז לעשות בניהם אינטרפולציה. מתקבלת שוב מטריקה חוקית ואפשר פורמלית להוכיח את זה.

וגם (לא ידוע איך מוכחים את זה) אבל מטריקה חדשה קובעת משטח תלת ממדי, בדרך כלל לא שטוח אשר מורכב מאותם קודקודים וקשתות כמו המשים שלנו ומרחקים בין כל נקודות הם כמו במטריקה הנוצרה מן אינטרפולציה.

אפשר לחשוב על זה ככה, שקיימת הזזה של נקודות של מש, תלת ממדית אשר תתן את מרחקים של מטריקה הנוצרה מתוך אינטרפולציה.

דוגמה מתוך המצגת של מאמר



המש שבאמצע (קשה לדמיין את זה) היא תלת ממדית ולא שטוח.

אבל וזה החדשנות של מאמר, במש בתלת ממדי הזה העיוות הזוויתי בין תמונת המקור (Source) לבין תמונת ביניים הוא סוּם על ידי העיוות הזוויתי בתמונת המטרה (Target)

## אינטרפולציה של מטריקות

הם נותנים הסבר די מסובך מתורה של מרחבים טופולוגיים על איך הם עשו את אינטרפולציה זאת, אבל אין לנו הרבה סיכוי להבין את זה באמת.

החישוב עצמו הוא מאד פשוט. עובר כל שני קודקודים, נחשב את מרחק בינם במש ראשון ( $L_1$ ) ומרחק בין אותם קודקודים במש שני ( $L_2$ ) ואז המרחק של אינטרפולציה יהיה כתלות בפרמטר  $t$

$$L_0 = \sqrt{t \cdot L_2^2 + (1-t) \cdot L_1^2}$$

עכשיו רואים שלזה יש כמה תכונות יפות שהם מגישות במאמר:

1. האינטרפולציה היא סימטרית (אם להחליף את שני תמונות וגם לתת  $t_1 = 1 - t$  אז נקבל אותה תוצאה.

2. כמובן אם  $t=0$  נקבל מרחקים של תמונה ראשונה ואם  $t=1$  אז נקבל את מרחקים של תמונה שניה.

# יצירה של מטריקה הסופית (על ידי מאמר ה-CETM)

פה בעצם נגמר התחום של מאמר של מירי.

הם משתמשים באלגוריתם CETM שהוכיחו ופיתחו לפני כמה שנים אשר יכול לקחת מטריקה שקיבלנו ולייצר מטריקה חדשה אשר כבר מישורית.

בשפה של משים שלנו זה אומר שהקלט של CETM הוא מרחקים שמצאנו אחרי אינטרפולציה והפלט הוא מרחקים חדשים אשר עבורם קיים סידור של נקודות של מש במישור עם מרחקים הללו.

גם מאחורי זה יש מתמטיקה די כבדה שאני גם לא ממש מבין. לצורך שלנו קל לדמיין שאלגוריתם בעצם פועל באותו אופן בו אנו היינו משטיחים משהו שדי מישורי אבל לא בדיוק מישורי. על ידי כך שהיינו לוחצים על נקודות ומנסים עד ידי כוח לגרום להשטיח את עצמו (למשל פיסה של נייר מקומטת).

פורמלית עבור כל קדקוד של מש (או במקרה רציף עבור כל נקודה בכלל) מגדירים מספר אשר **כופל** את מרחקים ממנו לכל הנקודות שכנות במספר **חיובי** (גדול או קטן מ1 כדי בהתאמה לייצג הגדלה או הקטנה של מרחק זה)

או פורמלית מגדירים פונקציה  $u(x)$  המוגדרת על כל נקודה המקבלת מיקום ומחזירה מספר כלשהו ואז (מתמטיקאים אוהבים את זה לצערי :- ) מגדירים פונקציה חדשה  $g = e^{u(x)}$  שבגלל תכונה של אקספוננט תמיד חיובית עבור כל פונקציה  $u(x)$ .  $g$  זה גם פונקציה כלשהי אבל תמיד חיובית עבור כל קלט.

## חישוב של אורכים של קשתות של מש

הרעיון הוא שאם (ובמקרה רציף זה קורה עבור אזור שמאד קטן) עבור כל נקודה נכפיל את מרחקים ממנה לשכנים שלה במספר קבוע אז לא נוסיף עיוות זוויתי.

הם גם הוכיחו שתחת תנאים ש  $u=0$  בקדקודים של גבול קיימים ערכים של  $u$  אשר נותנים מש מישורי.

עבור כל קדקוד של מש מגדירים משתנה  $u$  אשר ישפיע על אורכים של קשתות הנוגעות בקדקוד ככה שאחרי הפעלה של משתנים הללו נקבל אורכים חדשים.

כאשר נמצא בסוף ערכים מתאימים של  $u$ ים נקבל מרחקים המתאימים למש מישורי.

הנוסחה שעל פיה מחושב אורך החדש  $L$  של קשת לפי אורך הישן (שהתקבל מתוך אינטרפולציה) היא

$$(1) \quad L = L_0 \cdot e^{\frac{u_1}{2}} \cdot e^{\frac{u_2}{2}}$$

כלומר על אורך של כל קשת תשפיע  $u$  של שני הצמתים שבניהם הקשת מחוברת.

עבור צמתים שנמצאים בגבול של מש, נקבע  $u=0$  כלומר הם לא ישפיעו על קשתות הנוגעות בנו ובפרט קשתות של boundary בכלל יישארו באותו אורך כי הם מחברים את שני צמתי של boundary.

## פונקציה E וחישוב של נגזרות שלה

עכשיו הם מגדירים פונקציה  $E(\bar{u})$ , כלומר פונקציה של כל ערכי  $u$  אשר מאד מסובכת ולא צריך לדעת אותה. בחרו אותה ככה שהגרדיאנט שלה יתאפס כאשר המש מישורי. (כל נגזרת חלקית לפי כל  $u$  תתאפס)

גרדיאנט

אבל נגזרת חלקית של פונקציה זאת לפי משתנה  $u$  כלשהו היא פשוט  $2\pi$  פחות סכום של זוויות סביב קדקוד שאליו  $u$  הזה מתאים **ואז חלקי 2** כלומר היא מתאפסת כאשר סכום זוויות סביב כל קדקוד פנימי הוא  $2\pi$  ואז מש הוא מישורי. לוקטור של נגזרות חלקיות של פונקציה קוראים **gradient** והוא מתאפס כאשר מש מישורי.

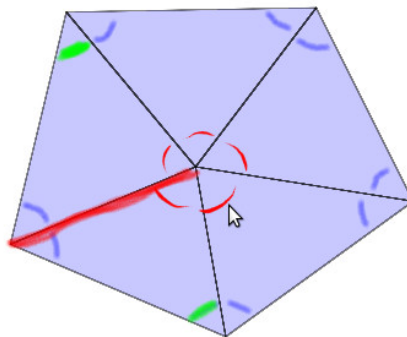
$$\frac{\delta E}{du} = \frac{2\pi - \text{sum of inner angles around vertex that corresponds to } u}{2} = \pi - \frac{\text{sum of inner angles around vertex that corresponds to } u}{2}$$

*hessian*

בנוסף הם גם מצאו את "נגזרת" שניה של פונקציה זאת שזה בשפת חדווה **hessian**. זהו מטריצה חאח של **נגזרות חלקיות מסדר 2** של  $E(\bar{u})$  לפי זוג של  $u$ 'ים. לפי נוסחאות שלהם (שאפשר לבדוק בקלות יחסית בסופו של דבר) מתקבל דבר הבא: הזוג של  $u$ 'ים מתאים לשני קודקודים פנימיים. כי אצלי בכל החישוב יש התעלמות מקדקודים חיצוניים כי  $u$  שלהם תמיד שווה 0.

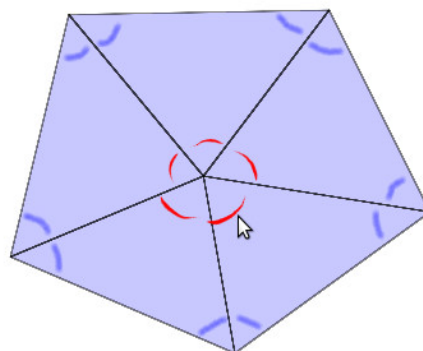
1. עבור  $u_1, u_2$  אשר קודקודים שלהם אינם שכנים, הנגזרת חלקית מסדר 2 זאת היא 0
2. עבור  $u_1, u_2$  אשר קודקודים שלהם כן שכנים, הנגזרת חלקית היא **מינוס** סכום של  $\cot$  של שני זוויות אשר מול הקשת שמחברת אותם **חלקי 4** (זוויות מסומנים בירוק):

$$Hessian[u_1][u_2] = - \frac{\cot(\alpha_1) + \cot(\alpha_2)}{4}$$



כאן באדום מסומנת הקשת שמחברת קודקודים שמתאימי ל  $u_1, u_2$  ובירוק מסומנים שני זוויות אשר סכום  $\cot$  של שניהם קובע את נגזרת חלקית.

3. ועבור מקרה שבו  $u_1 = u_2$ , כלומר נגזרת חלקית לפי אותו משתנה פעמיים, הערך הוא סכום של כל  $\cot$  של זוויות "חיצוניות" **חלקי 4**. הזוויות חיצוניות הכוונה ל כל זוויות המסומנות בכחול:



$$Hessain[u_1][u_1] = \frac{\text{sum of cots of all outer angles}}{4} = -(\text{sum of all other elements in row of } u_1)$$

לסיכום נשים לב לדבר הבא:

ההסיאנה הזו מטריצה שבה מופעים נגזרות חלקיות מסדר 2.

לדוגמה:

$du_1u_1$	$du_1u_2$	$du_1u_3$
$du_2u_1$	$du_2u_2$	$du_2u_3$
$du_3u_1$	$du_3u_2$	$du_3u_3$

אם ניקח למשל שורה 1, אז בה תופיעו נגזרות חלקיות לפי  $u_1$  ואז לפי  $u_1, u_2, u_3$  כלומר אם ניקח קדקוד אשר מתאים ל  $u_1$  אז נוכל למלא את שורה 1 על ידי ידע על שכנים של קדקוד זה וזווית חיצונית כמו שרשמתי למעלה.

1. בכל תאים המתאימים לטים אשר אינם שכנים לקדקוד של  $u_1$  יהיה 0 לדוגמה אם אין קשת  $u_1 - u_2$  אז בתא  $du_1u_2$  יהיה 0.
2. בכל תאים המתאימים לטים שכנים ל  $u_1$  יהיה מינוס סכום של  $\cot$  של זוגות זווית כמו שרשמתי למעלה
3. ובתא  $du_1u_1$  יהיה מינוס סכום של שאר תאים בשורה זאת.

הדבר כמובן נכון לגבי כל שאר שורות.

## חישוב של זווית של משולשים

- לצורך חישוב של gradient ושל hessian אנו בעצם צריכים עבור כל זווית שני דברים.
1. הערך שלו כדי לחשב את gradient (כי מבוסס על ערכים של זווית פנימיות)
  2. הערך של cot שלו כי עליהם מבוסס חישוב של ערכים של hessian.

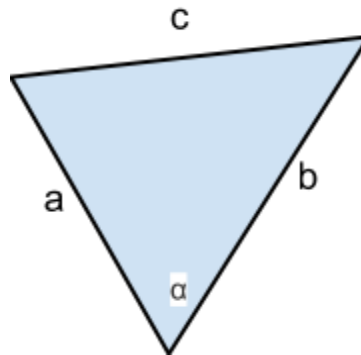
בגלל כמה שיקולים שבמבט לאחור אולי לא יצאו מספיק מוצדקים  
בחרתי לעשות את חישוב של שני הדברים הללו בדרך הבאה:

1. קודם כל אני מחשב את  $\tan(\frac{\alpha}{2})$  עבור כל זווית במש על ידי נוסחה הבאה שמופיעה במאמר ה-CETM:

$$\tan\left(\frac{\alpha}{2}\right) = \sqrt{\frac{(b+c-a)(c+a-b)}{(a+b-c)(a+b+c)}}$$

הנוסחה זאת שקולה למשפט קוסינוסים האומר כי:

$$c^2 = a^2 + b^2 - 2ab \cdot \cos(\alpha) \Rightarrow \cos(\alpha) = \frac{a^2 + b^2 - c^2}{2ab}$$



כאשר משולש הוא:

השתמשתי בנוסחה זאת מכמה סיבות: קודם כל חשבתי שאולי היא יותר יציבה נומרית ממשפט קוסינוסים רגיל.  
וגם כי ניתן להשתמש בה לחישוב של cot בלי להפעיל עוד פונקציה טריגונומטרית בדרך הבאה:

2. בחישוב של gradient אני משתמש בערכים של  $\tan(\frac{\alpha}{2})$  שחישבתי מראש עבור זווית פנימיות. עבור כל ערך כזה אני מפעיל עליו את atan וזה נותן לי את  $\frac{\alpha}{2}$  עבור כל זווית פנימית. את ערכים הללו אני מסכם ובסוף לוקח  $\pi$  ומחסיר ממנו את סכום שתקבל. זה הערך של גרדיאנט עבור קדקוד פנימי שעכשיו טיפלתי בו.
3. עבור חישוב של hessian אני שוב לוקח את ערכים של  $\tan(\frac{\alpha}{2})$  שחישבתי מראש אבל הפעם של זווית חיצונית (זוויות הללו היו פנימיות במשולשים אחרים קודם...)  
ומהם מחשב את  $\cot(\alpha)$  על ידי זהויות טריגונומטריות הבאות:

$$\tan(\alpha) = \frac{2\tan\left(\frac{\alpha}{2}\right)}{1-\tan\left(\frac{\alpha}{2}\right)^2}$$

ובנוסף מתקיים כי

$$\cot(\alpha) = \frac{1}{\tan(\alpha)}$$

ולכן:

$$2 \cdot \cot(\alpha) = \frac{1-\tan\left(\frac{\alpha}{2}\right)^2}{\tan\left(\frac{\alpha}{2}\right)}$$

וככה עבור זוג של זוויות  $\alpha_1, \alpha_2$  חיצוניות מצאתי את  $2\cot(\alpha_1)$  ואת  $2\cot(\alpha_2)$  ועבור hessian אני צריך  $\frac{\cot(\alpha_1)+\cot(\alpha_2)}{4}$   
ולכן אני מחשב את  $\frac{2\cot(\alpha_1)+2\cot(\alpha_2)}{8}$  במקום (חוסך כמה פעולות חילוק)

## חיפוש של ערכי ה-המתאימים על ידי שיטה של ניוטון

נשתמש בשיטה של ניוטון שעליה נלמד לצערי הרבה טוב טוב בנומריית אז אין צורך להסביר פה כמו שצריך...  
ערכים התחלתיים של שיטה במקרה שלנו יכולים להיות למשל:

$$\overline{u}_0 = \{0, 0, 0, \dots\}$$

ערכים התחלתיים נבחרו פחות או יותר סתם, במאמר שלהם הם מוכיחים שכל ערך התחלתי יהיה בסדר כי יש לפונקציה רק מינימום אחד גלובלי ומטריצה Hess מוגדרת חיובית (וכל זה לצערי הרב תלמדו בנומריית...): (-)

השיטה עצמה היא:

$$\overline{u}_{n+1} = \overline{u}_n - \nabla E(\overline{u}_n) \cdot Hess(\overline{u}_n)^{-1}$$

אבל כדי לא להפוך את מטריצה במקום כופלים את שני צדדים של שיטה ב  $Hess(u_n)$  ומקבלים:  
(המטריצה של Hessain היא מטריצה דלילה ולכן פחות יעיל להפוך אותה - תלמדו את זה בנומריית...): (-)

$$Hess(\overline{u}_n) \cdot \overline{u}_{n+1} = Hess(\overline{u}_n) \cdot \overline{u}_n - \nabla E(\overline{u}_n)$$

זה מערכת של משוואות לינארית ( $Ax=b$ ) כי בהינתן  $u_n$  ניתן לחשב את כל צד ימני ובצד שמאלי יש לנו מטריצה Hess ווקטור נעלם ה  $u_{n+1}$

### כמה הערות על שיטה של ניוטון

פשוט דיברתי על זה עם מירי אז אציין פה

השיטה של ניוטון שהשתמשנו בה לא חייבת להתכנס ובגלל זה בדרך כלל בעולם האמתי משתמשים בשיטות יותר מורכבות שבינם יש את trust zone region ויש את line search. לא לגמרי הבנתי מה הולך שם, ורק הבנתי שצריך שם גם לחשב את ערך של פונקציה ועל ידי זה לשנות קצת את התנהגות של שיטה כדי לא להגיע למצב שהשיטה "רוקדת" סביב הפתרון ולא מגיעה אליו כי עושה צעדים גדולים מידי.  
בטח גם על זה תלמדו בנומריית....



## ייצוג של hessian ופתרון של מערכת משוואות על ידי ספריית cholmod

המטריצה Hessian נשמרת בייצוג דליל בצורה הבאה:

1. מערך אחד גדול השומר את כל איברים השונים מ 0 כאשר איברים של עמודה ראשונה נשמרים לפי הסדר שלהם מהעליון לתחתון ואז איברים של עמודה הבאה ועד הסוף (למה עמודות אני לא יודע אבל זה מה שספרייה cholmod רוצה ממני, ובטח תלמדו על זה בנומריית גם כן) בפועל בגלל סימטריות אני מחשב את שורות ופשוט משקר ל cholmod שזה עמודות (לא אני עשית את זה אלא זה כבר היה בקוד של KVF)
2. מערך באותו גודל כמו מערך הראשון אשר עבור כל איבר בו אומר מה השורה שלו.
3. מערך של אינדקסים לתוך מערך הראשון אשר אומרים איפה מתחילה כל עמודה.

המחלקה אשר מנהלת את מידע זה נקראת אצלי CholmodSparseMatrix והיא יושבת ב cholmod\_matrix.cpp והיא התוצאה של refactoring רציני שעשית ל simplematrix שהיה אצלם. במחלקה יש פונקציה getCholmodMatrix שמחזירה מטריצה דלילה בדיוק בפורמט ש cholmod רוצה ממני.

עכשיו נשים לב שמתקיימים שני דברים:

1. מטריצה היא סימטרית
2. מטריצה היא מוגדרת חיובית (זה הם הוכיחו במאמר)

ולכן ניתן לפתור מערכת משוואות שהיא קובעת על ידי פירוק cholesky אשר גם עליו תלמדו בנומריית (אני גם הייתי רוצה ללמוד את זה... :-)

הפתרון של מערכת משוואות נעשה ב 3 שלבים:

1. cholmod\_analyze - זה פונקציה שעושה אנליזה ראשונית של מטריצה ואפשר למחזר את תוצאות שלה כל עוד מטריצה בעלת אותו מבנה כלומר האיברים השונים מ 0 נשארו באותם מקומות.
2. cholmod\_factorize - מבצעת את פירוק cholseky של מטריצה לשני מטריצות. ושומרת את תוצאה בפלט ביניים
3. cholmod\_solve - פותרת את מערכת משוואות על ידי פירוק שמצאה cholmod\_factorize

# סקיצה של אלגוריתם הכולל

חשוב להדגיש שבכל מקום בקוד משום מה קראתי לטים בשם Kים.

## הזיכרון של אלגוריתם:

1. מערך L0: - אורכים של כל קשות שהתקבלו אחרי אינטרפולציה. מחושב רק פעם אחד התחלה. יש לי שיטה שלפיה אני יודע באיזו תא של מערך יש איזו אורך.
2. מערך K: פה נשמרים כל Uים של קודקודים פנימיים. גם פה אני יודע באיזו תא יש איזו U.
3. מערך L: כאן נשמרים אורכים של קשות אחרי שהפעילו עליהם את Uים. מחושב בכל איטרציה של שיטה של ניוטון. איברים נשמרים באתו סדר כמו ב L0
4. מערך mem: שם מבלבל - כאן נשמרים זמנית tan'ים של חצי הזווית של כל משולשים של מש.

## האלגוריתם הוא:

פונקציה ראשית של אלגוריתם היא interpolate\_frame מבצעת את דברים הבאים:

- אתחול  
הצב 0 בכל איברים של מערך K (אני בקוד קורא למשתנים U בשם K).  
אני לפעמים לא עושה את זה לצורך של אופטימיזציה קטנה, אם ממש לא מזמן כבר הרצתי את אלגוריתם עם ערכי דומים. אז אני מאמין שעדיף להתחיל את חיפוש עם ערכי K שאז מצאתי.
- אינטרפולציה של מטריקה ( קריאה ל metric\_create\_interpolated שעושה):  
בהינתן שני משים a,b, חשב את אורכים של קשות בשני משים ואז אורכים חדשים קשות ושים את תוצאה במערך L0  
(לגבי איך נדע איזו קשת נשמור באיזו אינדקס במערך בהמשך).
- איטרציות של ניוטון בשביל השטחה של מטריקה (בעצם CETM): (קריאה ל metric\_flatten שעושה):  
בצע 50 פעמים (סתם מספר שאחריו אני מכריז שאין התכנסות של שיטה):

1. חישוב של gradient ושל hessian (קריאה לפונקציה: calculate\_grad\_and\_hessian שעושה):  
בצע:

(a) חשב אורכים חדשים L על ידי אורכים L0 ומקדמים K (כל פעם שצריך K של קדקוד חיצוני תשתמש ב0) לפי נוסחה (1)

(b) הכנה לחישוב של זוויות:

לפי אורכים חדשים של L חשב את  $\tan(\frac{\alpha}{2})$  (פונקציית עזר שעושה את זה היא: calculate\_tan\_half\_angle)  
(פונקציה גם יודעת להתמודד עם מקרים בהם 3 אורכים אינם מקיימים את אי שוויון של משולש ומחזירה 0 או אינסוף בהתאמה כאילו שמשולש הוא מנוון) (רשום לעשות את זה במאמר, מניסיון שלי כאשר מתקבלים משולשים הללו בדרך כלל שום דבר לא מתכנס אז זה לא ממש עוזר...)  
ותוצאה נשמרת במערך mem (שם מבלבל)

(c) חישוב של גרדיאנט וחישוב של נורמה של גרדיאנט:

עבור כל קדקוד פנימי היא מחשבת atan של זוויות פנימיות על פי ערכים של  $\tan(\frac{\alpha}{2})$  אשר שמורים בממ בתאים המתאימים (עוד שניה אני אסביר איך אני יודע איזו ערכים לקחת), מסכמת אותם, רושמת איבר מתאים בגרדיאנט (המשתנה נקרא EnergyGradient) וגם מוסיף, את ריבוע של ערך זה לסכום (ככה אני בו זמנית מחשב את נורמה של גרדיאנט)

d. חישוב של הסיאן:

חשב את cot של זווית חיצונית על פי גם ערכים של  $\tan(\frac{\alpha}{2})$  אשר שמורים בממ ומלא לפי הסדר את שורה ב Hessians (אצלי נקרא EnergyHessian) (אני רושם שם רק את איברים עד אלכסון הראשי אבל מחשב את כל איברים כי

אני צריך אותם כדי לחשב את איבר באלכסון).

2. בדיקה של תנאי יציאה:

בדוק מה יצאה הנורמה L2 של גרדיאנט (כלומר מדד שאומר עד כמה הוא קרוב ל0)  
(נשמר במשתנה grad\_norm בפונקציה calculate\_grad\_and\_hessian)  
אם מספר זה קטן מן END\_ITERATION\_VALUE שאני סתם בחרתי כ  $1 \cdot 10^{-10}$

אז **עצור את איטרציות.**

בפועל הדיוק בר השגה (עוד קללה מנומרת) פה הוא בערך  $1 \cdot 10^{-14}$  (בדקתי ניסיונית)  
אבל כדי קצת להאיץ דברים כי לא מרגישים הבדל בחרתי במספר זה

3. הכנה לפתרון של מערכת משוואות עבור שיטה של ניוטון:

חשב את צד ימני של משוואה (המשתנה NewtonRHS כלומר newton right hand side) לפי  
ערכים קיימים של K, של גרדיאנט ושל hessian.  
המשתנה NewtonRHS זה בפועל סתם מערך אבל עטפתי אתו במחלקה CholmodVector כדי קצת  
לעשות דברים יותר יפים.  
(בפרט אפשר להכפיל את CholmodMatrix ב CholmodVector)

4. פתרון של מערכת משוואות דלילה: פתור את ממערכת משוואות עם

עם cholmod - התוצאה מתקבלת במשתנה Xcholmod) ושם את תוצאה בK

5. חזור חזרה להתחלה של לולאה.

• **קביעה של מיקומים חדשים של קודקודים (פונקציה metric\_embed)**

בצע:

1. קביעה של תנאים נוספים כדי לקבל פתרון יחיד:

בחר קשת E0 (נבחרת רק פעם אחד כאשר טוענים את מודל יותר נכון)  
בצע אינטרפולציה לינארית על מיקום של קדקוד ראשון שלה בין שני משים ,  
ובצע אינטרפולציה לינארית על כיוון שלה בשני משים ועל ידי זה  
ועל ידי אורך שלה שנקבע על ידי מטריקה קבע את מיקום של קדקוד שני שלה  
וסמן ששני קדקודים הללו קיבלו את מיקום שלהם  
2. כל עוד יש קודקודים שלא קיבלו את מיקום שלהם:

a. בחר קדקוד שיש לו שני קודקודים שקיבלו את מיקום כבר

b. ועל ידי מיקום של שני קודקודים הללו אורכים של צלעות קבע את מיקום שלו

c. וסמן שהוא קיבל את מיקום.

השלב זה נעשה בערך על ידי DFS מן אחד הקדקודים של קשת E0.

# הסבר יותר מפורט על מימוש של אלגוריתם

השיטה של קומפילציה של חישוב שהשתמשתי

עכשיו אענה על שאלה איך אני יודע איזו איברים לקחת מכל אחד המערכים, איך לדעת מי השכנים של כל קדקוד, ועוד ועוד ועוד.

בנוסף למערכים שציינתי יש לי עוד 3 מערכים של פקודות (מן **bytecode שאלתרתי**). אשר נבנים פעם אחד בזמן טעינה של מודל (על ידי פונקציה initialize וגם מחלקה BDMORPH\_BUILDER שנקראת מפונקציה זאת)

- `init_cmd_stream`
- `iteration_cmd_stream`
- `extract_solution_cmd_stream`

מערכים הללו מכילים מן פקודות "אסמבלי" שהמצאתי לצורך חישוב ואני בונה אותם פעם אחד כאשר מתבצעת טעינה של מודל והסיבה לכך היא שמודל נטען רק לעתים רחוקות ואינטרפולציה נעשית המון פעמים.

**המערך `init_cmd_stream` (מכיל רק בעצם קשתות של גרף)**

פה זוגות של `double`ים כאשר כל זוג מייצג קשת בגרף.

`metric_create_interpolated` עוברת על זוגות הללו ולפי סדר הופעתם מחשבת את איברים בL0

לכן `metric_create_interpolated` נראית ככה: (הקוד האמתי קצת יותר ארוך בגלל הערות וasserts):

```
void BDMORPHModel::metric_create_interpolated() {
    CmdStream commands(*init_cmd_stream);
    int edge_num = 0;
    while (!commands.ended()) {
        uint32_t vertex1 = commands.dword();
        uint32_t vertex2 = commands.dword();

        double dist1_squared =
            modela->vertices[vertex1].distanceSquared(modela->vertices[vertex2]);
        double dist2_squared =
            modelb->vertices[vertex1].distanceSquared(modelb->vertices[vertex2]);

        double dist = sqrt((1.0-current_t)*dist1_squared+current_t*dist2_squared);
        L0[edge_num++] = dist;
    }
}
```

כלומר מתוך `init_cmd_stream` אני קורא זוגות של `dwords` (זה 4 בתים) (בתוך `CmdStream` יש מצביע פנימי שמתקדם כל פעם שקוראים ממנו משהו) כמו קובץ אפשר להגיד.

## המערך `iteration_cmd_stream` (מכיל פקודות הקובעים את הריצה של פונקציה המחשבת את הגרדיאנט ואת ההסיאן):

הרעיון המרכזי פה הוא שהמערך של פקודות הזה מכיל הרבה פקודות מ-3 סוגים הנ"ל שנמצאים שם אחד אחרי השני בערבוב:

כלומר כמה פקודות שמחשבים אורך של קשתות, אחריהם כמה פקודות שמחשבים זוויות, אחריהם אולי עוד כמה פקודות שמחשבות אורך של קשתות, פקודה שמטפלת בקדקוד (מחשבת איבר בגרדיאנט ושורה בהסיאן על ידי תוצאות של פקודות קודמות) ועכשיו עוד קצת פקודות שמחשבות אורכים וככה הלאה.

אני מערבב את פקודות כדי שכל פקודה תשמש בפלטים של פקודות שהתבצעו לא מזמן, ותוצאות שלהם עוד ב-`cache`.

### היצירה של פקודת עבור `iteration_cmd_stream`

היצירה של פקודת נעשית בערך באופן הבא:

אני בוחר צומת וממנו עושה BFS על כל הגרף. וכל צומת פנימי שאני פוגש אני "מטפל" בו. כלומר מטרה היא בסופו של דבר לחשב עבורו את תא בגרדיאנט ושורה בהסיאן

בשביל זה אני צריך לדעת את ערכים של כל זוויות פנימיות וחיצוניות. אני בודק מי מהם כבר חושב על ידי פקודת `COMPUTE_HALF_TAN_ANGLE` שכבר נמצאת במעריך. ומשתמש בתאים הללו. כל זוויות שעדיין לא חושבו גורמים להוספה פקודת `COMPUTE_HALF_TAN_ANGLE` למעריך. ובסוף אני כאשר אני יודע שיצרתי כל פקודת `COMPUTE_HALF_TAN_ANGLE` עבור כל זווית הפנימיות וחיצוניות אני כותב פקודה `COMPUTE_VERTEX_INFO` ואחרי כתובות של תאים בממ `mem` שבהם יושבים תוצאות של פקודות הללו.

באותו אופן, כל פעם שרוצים להשים פקודה `COMPUTE_HALF_TAN_ANGLE`, לפניה חייבים להופיע פקודות `COMPUTE_EDGE_LEN` עבור כל צלע של משולש.

לכן קוד אשר יוצר את פקודה `COMPUTE_HALF_TAN_ANGLE` שם לפניה פקודות `COMPUTE_EDGE_LEN` עבור כל קשתות שעוד לא חושבו.

אחרי שהסתיים טיפול בקדקוד, אני מטפל באותו אופן בשכנים שלו שעדיין לא ביקרתי בהם, כאשר אני ממשיך להוסיף פקודת `COMPUTE_HALF_TAN_ANGLE` ו-`COMPUTE_EDGE_LEN` עבור כל זווית או קשת חדשה שאני פוגש בדרך (כלומר כזאת שעוד לא חישבתי את אורך או ערך שלה לפני, על ידי הפקודות הללו).

הסיבה שעשיתי מערך פקודות כזה היא שה-`mesh` בתוכנה שלנו נשאר קבוע לאורך כל הריצה. כל מה שמשתנה זה מיקומים של קדקודים. לכן במקום כל פעם לעשות טיול עליו, לגלות מי הם זוויות פנימיות וחיצוניות, להשתמש במבנים אטיים כדי לשמור בהם מידע על זוויות ואורכים (למשל כדי לדעת זווית לפי 3 קודקודים הייתי אמור להשתמש בעץ חיפוש בינרי עם מפתח של שלושה של קדקודים).

במקום לעשות כל זה כל פעם, אני עושה את כל העבודה מראש, ושופך את תוצאות שלה למעריך פקודות שאפשר לבצע באופן עיוור בלי לדעת כלום לא על מבנה של הגרף.

איך זה נשפך? כל פקודה מקבלת את הפרמטרים שלה מראש ככה שלא צריך בהמשך לחשב מחדש כל פעם את הפרמטרים.

כל פקודה משתמשת בתוצאות של פקודה קודמת (פרט לפקודה `COMPUTE_EDGE_LEN`) אשר משתמשת ב-`L0` (שחושב כאשר עשינו אינטרפולציה של מרחקים `Ki` אשר בהתחלה הוא 0 ואחרי הוא פלט של איטרציה קודמת) הפלט של פקודת ביניים נכתב למעריך הפלט המתאים, תמיד לתא "האחרון" והוא מתקדם. כלום אפשר להגיד שפלט מתווסף (`appends`) לסוף של מעריך.

מהערכי פלט ביניים הם `L` (והסוף הנוכחי שלו נקבע על ידי משתנה `edge_num`)

ומערך mem אשר במקור היה צריך לשמור את כל פלט ביניים אבל הוא לא בסוף. זה משתנה מטיפוס TmpMemory - מחלקת עזר קטנה שגם היא מאפשרת להוסיף שם לסוף פלט.

## הפקודות שהמערך מכיל הם:

1. פקודה COMPUTE\_EDGE\_LEN:  
מיד אחריה באים כפרמטרים אינדקסים של שני אים מתוך מערך Ka  
פקודה מחשבת אורך של קשת הבאה בL על ידי אורך L0.
2. פקודה COMPUTE\_HALF\_TAN\_ANGLE:  
מיד אחריה באים אינדקסים של 3 אורכים מתוך L, (נגיד צלע a,b,c)  
פקודה מחשבת את  $\tan(\frac{\alpha}{2})$  של זווית בין קשת a,b מול ושמה את תוצאה באיבר הבא במem.
3. פקודה COMPUTE\_VERTEX\_INFO:  
מיד אחריה באים:
  - האינדקס Ka של קדקוד,
  - מספר השכנים שלו
  - אינדקסים בתוך mem של זווית הפנימיות שכבר חושבו על ידי COMPUTE\_HALF\_TAN\_ANGLE
  - אינדקסים של תאים בתוך mem של זווית חיצונית יחד עם מה האינדקס Ka שלהם.על פי מידע זה מחשב את תא בתוך gradient ושורה בתוך hessian

הקוד של calculate\_grad\_and\_hessian אשר מבצע פקודות מן iteration\_cmd\_stream:

```
void BDMORPHModel::calculate_grad_and_hessian() {
    int edge_num = 0;
    CmdStream commands(*iteration_cmd_stream);
    TmpMemory mymem = mem;
    grad_norm = 0;
    EnergyHessian.startMatrixFill();

    while(!commands.ended()) {
        switch(commands.byte()) {
            case COMPUTE_EDGE_LEN: {
                int k1 = commands.dword();
                int k2 = commands.dword();
                L[edge_num] = edge_len(L0[edge_num], getK(k1), getK(k2));
                edge_num++;
                break;

            } case COMPUTE_HALF_TAN_ANGLE: {
                double a = L[commands.dword()];
                double b = L[commands.dword()];
                double c = L[commands.dword()];
                double tangent = calculate_tan_half_angle(a,b,c);
                mymem.addVar(tangent);
                break;

            } case COMPUTE_VERTEX_INFO: {
                Vertex vertex_K_num = commands.dword();
                int neigh_count = commands.word();
                int k_count = commands.word();

                /* calculate gradient */
                double grad_value = M_PI;
                for (int i = 0 ; i < neigh_count ; i++) {
                    double value = mymem[commands.word()];
                    double angle = atan(value);
                    grad_value -= angle;
                }
                EnergyGradient[vertex_K_num] = grad_value;
                grad_norm += (grad_value*grad_value);

                /* calculate row in the Hessian */
```

```

double cotan_sum = 0;
for (int i = 0 ; i < neigh_count ; i++)
{
    double twice_cot1 =
        twice_cot_from_tan_half_angle(mymem[commands.word()]);
    double twice_cot2 =
        twice_cot_from_tan_half_angle(mymem[commands.word()]);
    double value = (twice_cot1 + twice_cot2)/8.0;
    cotan_sum += value;

    if (i < k_count) {
        VertexK neigh_K_index = commands.dword();
        EnergyHessian.addElement(
            vertex K num, neigh K index, -value);
    }
    EnergyHessian.addElement(vertex K num, vertex K num, cotan sum);
}
}}
}
grad_norm = sqrt(grad_norm);
}

```

הדוגמה לצורך המחשה של iteration\_cmd\_stream נראה בערך ככה:

```

COMPUTE_EDGE_LEN (1 byte)
K index (4 bytes)
K index (4 bytes)
COMPUTE_EDGE_LEN (1 byte)
K index (4 bytes)
K index (4 bytes)
...
COMPUTE_HALF_TAN_ANGLE (1 byte)
L index (4 bytes)
L index (4 bytes)
L index (4 bytes)
COMPUTE_HALF_TAN_ANGLE (1 byte)
L index (4 bytes)
L index (4 bytes)
L index (4 bytes)
COMPUTE_HALF_TAN_ANGLE (1 byte)
L index (4 bytes)
L index (4 bytes)
L index (4 bytes)
COMPUTE_EDGE_LEN (1 byte)
K index (4 bytes)
K index (4 bytes)
COMPUTE_EDGE_LEN (1 byte)
K index (4 bytes)
K index (4 bytes)
COMPUTE_VERTEX_INFO (1 byte)
vertex K index (4 bytes)
neighbour count (1 byte)
K count (1 byte)
TMP index
TMP index
...

```

# הסבר יותר מפורט על קביעה של מיקום של קודקודים (embedding של מטריקה בשפה שלהם)

אחרי שאיטרציות של ניוטון התכנסו, יש לנו מטריקה שטוחה ורק נשאר למצוא את מיקומי של קודקודים.

## בחירה של פתרון יחיד

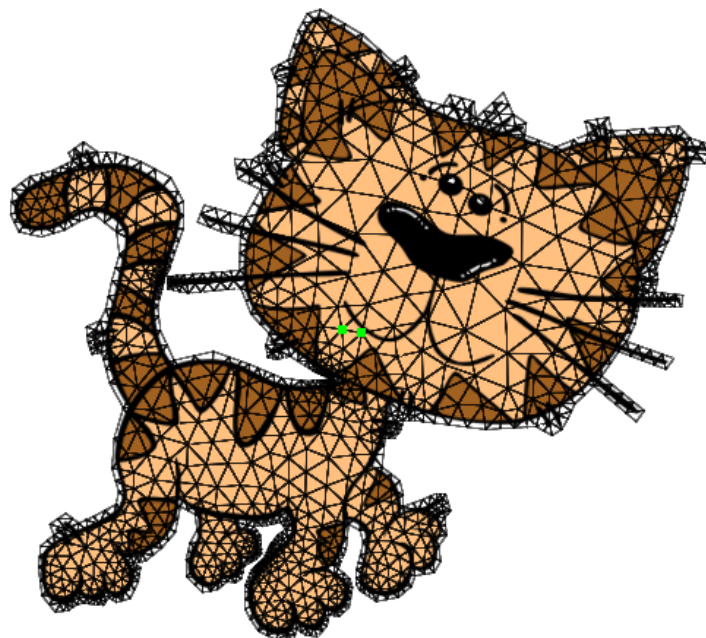
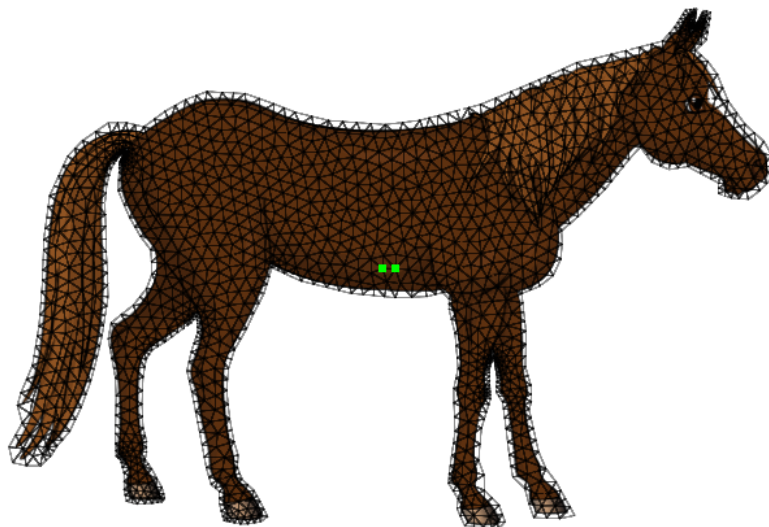
נשים לב שיש עדיין אינסוף פתרונות כי אם נניח שמצאנו סט אחד של מיקומים נוכל לסובב ולהזיז אותו בלי לשנות את מרחקים.

במאמר רק מזכירים את זה בלי לתת פתרון של מה לעשות.

התמודדתי עם בעיה זאת בדרך הבאה:

בחרתי קודקוד הכי קרוב למרכז של המש הראשון ולידו בחרתי שכן שלו אקראית (סתם שכן ראשון שראיתי) זה נותן לי את קשת התחלתית.

התוצאות יצאו די טובות:





את מיקום של אחד הקדקודים של קשת זאת אני מוצא על ידי אינטרפולציה לינארית בין שני משים כתלות ב גם את כיוון שלה אני מוצא על ידי אינטרפולציה לינארית בין שני משים.  
ואז את קדקוד שני שלה אני שם במרחק שמצאתי על ידי אלגוריתם BDMORPH ובכיוון המתאים.

הקוד שעושה את זה הוא: (קשת התחלתית היא  $e_0$ )  $edge1\_L\_location$  | זה מיקום של אורך של קשת זאת ב  $L$

```
Vector2 e0_direction1 = (modela->vertices[e0.v1] - modela->vertices[e0.v0]).normalize();
Vector2 e0_direction2 = (modelb->vertices[e0.v1] - modelb->vertices[e0.v0]).normalize();

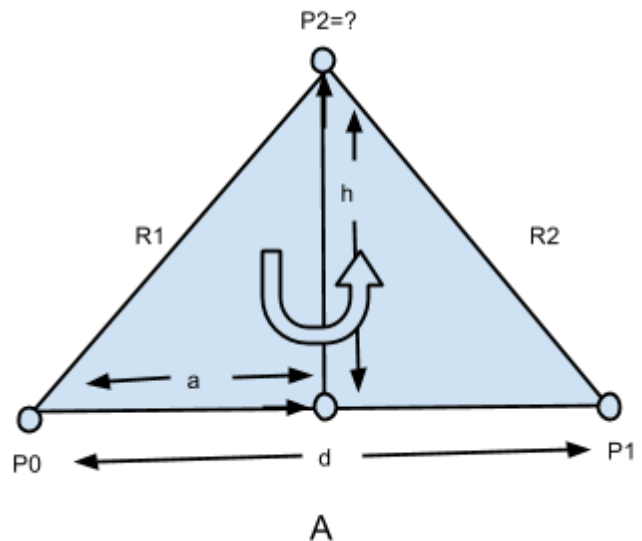
Vector2 e0_direction = (e0_direction1 * (1.0-current_t) + e0_direction2 *
current_t).normalize();

vertices[e0.v0] = modela->vertices[e0.v0] * (1.0-current_t) +
modelb->vertices[e0.v0] * current_t;

vertices[e0.v1] = vertices[e0.v0] + e0_direction * L[edge1_L_location];
```

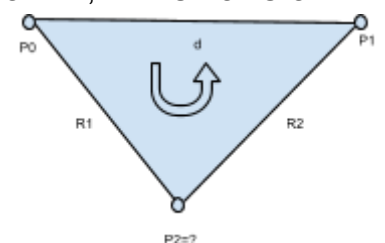
### חישוב של מיקום של קדוקודים - פירוט

אז עכשיו יש לנו מיקום של שני קדוקודים. את מיקום של שאר קדוקודים נמצא באופן רקורסיבי:  
בהינתן מיקום של שני קדוקודים ואורכים של צלעות המשולש ניתן למצוא את מיקום של קדוקוד שלישי:



נתון לנו מיקום של קדוקוד  $P_0$  ומיקום של קדוקוד  $P_1$  כמוכן ידועים לנו אורכים  $d$  ואורכים  $R_1$  ו  $R_2$ .  
 $hd$  בעצם זה נתון מיותר כי ניתן לחשב אותו ממרחק בין שני קדוקודים  $P_0$  ו  $P_1$  ואני אכן עושה את זה כי זה משום מה משפר את דיוק של האלגוריתם.

המטרה היא למצוא את מיקום של קדוקוד  $P_2$  האמת יש פה שני פתרונות, הזה שהצגתי למעלה שבו הקדוקוד הוא "מעל" והשני שהוא מתחת:



כדי להתמודד עם זה אני מניח שהסדר של קודקודים הוא תמיד כנגד כיוון של שרון  
 לכן במקרה הראשון אני מקבל את קודקודים בקלט בסדר  $P_0, P_1, P_2$  ובמקרה שני בסדר  $P_1, P_0, P_2$

כדי למצוא את מיקום של  $P_2$  אני מוצא את שני אורכים,  $a$  וה  $h$  (זה גובה של משולש)

1. את  $a$  מוצאים ככה:

מצד אחד מתקיים כי לפי משפט קוסינוסים:

$$R_2^2 = R_1^2 + d^2 - R_1 d \cdot 2 \cos(P_2 P_0 P_1) \Rightarrow$$

$$\cos(P_2 P_0 P_1) = \frac{R_1^2 + d^2 - R_2^2}{2 \cdot R_1 d}$$

אבל מהגדרה של  $\cos$  זה ניצב על יד חלקי יותר, כלומר

$$\cos(P_2 P_0 P_1) = \frac{a}{R_1}$$

נשווה שני משוואות ונכפיל ב  $R_1$  ונקבל:

$$a = \frac{R_1^2 - R_2^2 + d^2}{2 \cdot d}$$

2. עכשיו נמצא את  $h$  לפי משפט פיתגורס:

$$h = \sqrt{R_1^2 - a^2}$$

3. עכשיו נמצא את וקטור  $P_0 \rightarrow P_1$  על ידי חיבור של שני נקודות, ננרמל אותו, ונכפיל ב  $a$  ולזה נחבר את נקודה  $P_0$

כך קיבלנו נקודה בה גובה חותך את בסיס של משולש.

עכשיו ניקח את וקטור  $P_0 \rightarrow P_1$  המנורמל ונסובב אתו 90 מעלות נגד כיוון שרון,

ואז נקבל תמיד את וקטור בכיוון הגובה.

נכפיל אותו ב  $a$  ונחבר אותו למיקום של בסיס הגובה.. זהו קיבלנו מיקום של קדקוד הראש.

## חישוב של מיקום של קודקודים - אלגוריתם

החישוב שנעשה הוא:

1. חשב את  $R_1^2, R_2^2, d^2$  על פי אורכים במערך  $L$  שהתקבלו מאיטרציות של ניוטון

2. חשב את  $\frac{R_1^2}{d^2}$  ואת  $\frac{R_2^2}{d^2}$

3. חשב את

$$\frac{a}{d} = \frac{1}{2} \left( \frac{R_1^2}{d^2} + \frac{R_2^2}{d^2} \right) + \frac{1}{2}$$

4. חשב את:

$$\frac{h}{d} = \sqrt{\frac{R_1^2}{d^2} - \left(\frac{a}{d}\right)^2}$$

5. חשב את:

$$\bar{A} = (P_1 - P_0)$$

$$P_2 \cdot x = P_0 \cdot x + \frac{a}{d} \cdot \bar{A} \cdot x - \frac{h}{d} \cdot \bar{A} \cdot y$$

$$P_2 \cdot y = P_0 \cdot y + \frac{a}{d} \cdot \bar{A} \cdot y + \frac{h}{d} \cdot \bar{A} \cdot x$$