# Check if precompute is necessary, otherwise skip to ARAP

Before actually precomputing we first check if we actually need to precompute.
There are two things we check for to determine whether to precompute or skip to ARAP and use data stored by precompute in a previous run of the program.
1. **Dirty data**: Implies handles have been added, removed or adapted.
2. **Precomputed data exists**: All precomputed data is stored on the source object.
This data is stored as entries in a dictionary. If any of the entries that should exist in the dictionary after a precompute don't exist we must precompute. This is for example the case if you run the program and an error occurs in precompute. When you would rerun the program it would have the same handles, but non-existing data, resulting in errors in ARAP,which tries to find this data.

# Precompute

$$x' = \operatorname{argmin} \left| \underbrace{\begin{pmatrix} \sqrt{w_{12}} & -\sqrt{w_{12}} & & \\ \sqrt{w_{13}} & & -\sqrt{w_{13}} & \\ \sqrt{w_{14}} & & & -\sqrt{w_{14}} \end{pmatrix}}_{A} \overbrace{\begin{pmatrix} x_1' \\ x_2' \\ x_3' \\ x_4' \end{pmatrix}}^{x'} - \underbrace{\begin{pmatrix} \sqrt{w_{12}}\,(x_1 - x_2)\,R_1 \\ \sqrt{w_{13}}\,(x_1 - x_3)\,R_1 \\ \sqrt{w_{14}}\,(x_1 - x_4)\,R_1 \end{pmatrix}}_{b} \right|^2$$

## Data precomputed and stored in Precompute?

In precompute constant data is computed, which can then be used by ARAP.
The following data is precomputed:

1. **Amount of rows** for A and A', used as amount of rows for b in ARAP.
2. List containing the constant part of b: $\sqrt{w_{vn}}(x_v - x_n)$ **(sqrtWivTimesxvMinxi)** for each one ring with center vertex **v** and neighbour vertices **n**.
3. **Sparse packages** for A', A and the **LU decomposition** of $A'^T A'$

## What is Sparse Packing?

Sparse matrices take much less space in RAM, because they only store non-zero elements. Mostly sparse matrices are used in ARAP. As a consequence much larger mesh sizes can be deformed with ARAP. However sparse matrices could not be stored on the source object as this would crash blender, which would require us to store them as dense matrices.
This diminished the benefits of sparse matrices as larger meshes could no longer fit memory in their dense format. What's more, after each run of the program, blender would still use an abundance of RAM to keep the data stored. This would make the computer very slow afterwards. Only restarting blender could solve this issue. This is why we use a method we call Sparse Packing. This method works as follows:

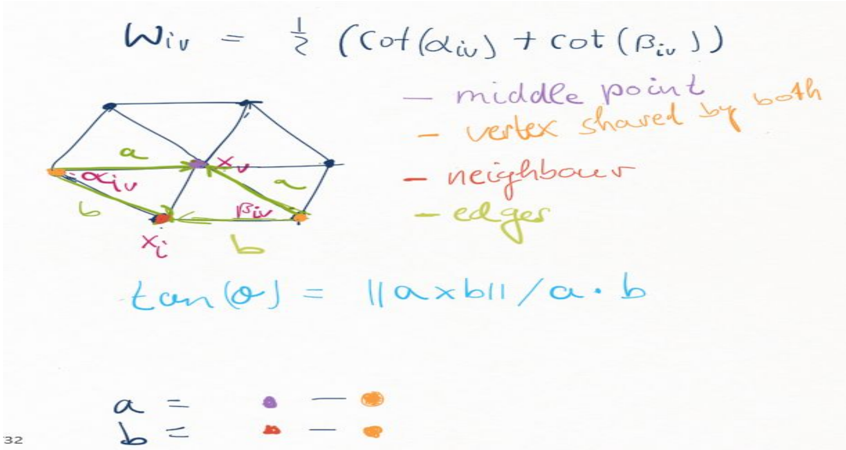
1. Using the scipy.sparse.find() we can get the rowIndex, columnIndex, value combinations of all non-zero values in a sparse matrix. This is then stored as a dense matrix. Since most values are zero, this massively reduces the required storage space.
2. In precompute we store sparse matrices using the function Package_SparseMatrix(). Given a sparse matrix, it stores the shape and a dense matrix containing the result of using scipy.sparse.find() on this sparse matrix.
3. In ARAP the shape, row indices, column indices and values are used to rebuilt the original sparse matrix. This is done by the function UnPackage_SparsePackage().

*Results*: Larger meshes can be deformed, much less RAM is required for storage.
*Disadvantage:* ARAP has to rebuilt the sparse matrices every time.
*Why?*: We want to have an implementation that wouldn't require to restart blender every time you run

the program, because your pc got extremely slow. We also want the implementation to be usable on larger meshes.

## How data is obtained

1. **Amount of rows**: The amount of rows of A and A’ is equal to the sum of one-ring neighbours for each vertex of the mesh. In order to get this amount the meshes are converted to bmesh. As a consequence the functions of the bmesh library can be used.
   Which can then be used to get the one-ring neighbours of each vertex. These are summed up to get the total amount of rows.
2. $\sqrt{w_{vn}}$ **(cotangent weights)** is required to compute the non-zero values in A, A’ and also to compute $\sqrt{w_{vn}}(x_v - x_n)$ , the constant part of b.
   The cotangent weights are computed by finding the matching neighbours of the middle vertex and a neighbour, see image below and
   https://in.answers.yahoo.com/question/index?qid=20110530115210AA2eAW1
   for more information.

   Flat meshes sometimes have $a \bullet b = 0$, since a and b are orthogonal.
   What’s more sometimes meshes are non-closes and may have only 1 of the two cotangents. In these cases a default weight of 1 is used. Also when weights are negative a very small value is used (e.g.) 10e-3 to avoid square root problems.
3. **A’ and A**:
   **A** can be seen as a rows x columns size matrix. Where rows are **Amount of rows.**
   The amount of columns is equal to the amount of vertices in the mesh, where each vertex index matches to a column index in A, so vertex indexed 5 matches column 5 in A.
   Each row contains two non-zero elements. A $\sqrt{w_{vn}}$ at the column index matching the center vertex of a one-ring and $-\sqrt{w_{vn}}$ at the column index matching a one-ring neighbor vertex of the center vertex. A is filled by looping over all the one-ring neighbours of the bmesh vertices.
   **A’:** In order to get A’ from A it is necessary to remove the constraint columns from A. However removing columns from a matrix requires the matrix to be rebuilt every time a column is deleted. This makes removing columns highly inefficiënt. Which is why we took a different approach:
   Let's say the mesh consists of 10 vertices then A would have 10 columns, where every row is filled with 2 non-zero values.
   Now let’s say the vertices with indices: 0,2 and 4 are constraint indices.
   Then A’ must have 10 - amount of **constraint indices** = 10-3 = 7 columns.
   So A’ would be a matrix with the columns that belong to the vertices indexed: 1,3,5,6,7,8,9.
   We then map the vertices indexed:1,3,5,6,7,8,9 to columns 0,1,2,3,4,5,6 of A’.
   This is done using a list containing **validIndices**. This is a list that contains all the non constraint indices. The list is sorted and doesn’t contain duplicates, such that it works with multiple handles.

Because it’s sorted we can ask for the index of a valid index in **valid Indices**.For example, if we would ask for a 3 in this example , we would get a 1 (the column index that matches to the vertex indexed 3).

In order to get the index of a vertex we also use a **KD-Tree** which links vertex coördinates to the respective vertex indices in the bmesh. Although an enumeration over bmesh gives us the indices of the center vertices of the one-rings, we don’t have the indices of the one-ring neighbours that’s why a KD-Tree is used.

4. **LU-Decomposition**: We basically use this example code, to decompose $A'^T A'$**:** http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.sparse.linalg.SuperLU.html

# ARAP

1. **Check for existing deformed:** One of the parameters passed to ARAP **existingDeformed**, Is a boolean that indicates whether to make an initial guess or not. If so a translation is applied on constraint vertices of deformed_mesh.
2. **Local step:** Next the local step is computed as is indicated in step 1.2, see: http://www.cs.uu.nl/docs/vakken/ddm/Practical%20Exercise%203%20-%20Mesh%20Deformation.pdf. In order to compute P and Q once again the inputted source and deformed meshes are converted to bmesh. All the Rotations are stored in a list: **Rvs**.
3. **b**’: All that remains now to solve for $A'^T A'x' = A'^T b'$ is to compute b’.

$$b' = b - A \overbrace{\begin{pmatrix} 0 \\ x'_{\text{CONST}} \end{pmatrix}}^{\text{zeroXPrimeConst}}$$

   As A has already been stored in precompute all that is necessary now is to compute **zeroXPrimeConst** and b.

   **b:** We loop over all the one-rings neighbours of the deformed mesh vertices. Furthermore a counter is used: **rowCount**, which keeps track of the row we’re working on. Also the list with $\sqrt{w_{vn}}(x_v - x_n)$ from precompute is retrieved.

   In the loop we retrieve the $\sqrt{w_{vn}}(x_v - x_n)$ and rotation from their list using **rowCount.** Then the respective row is filled in matrix b.

   **zeroXPrimeConst:** In the loop we check in the index belonging to the bmesh vertex is a constraint vertex. If it is, its coördinates are put in the respective row of the **zeroXPrimeConst** Matrix. If it is not, 0’s are put in as the coördinates.
4. The missing components of the **LU decomposition** are retrieved and xPrime is computed and the deformed mesh is adapted accordingly. Finally ARAP returns the deformed mesh.

# Iteration

In the main function the tolerance is obtained by getting the diagonal of the bounding box of source and taking 10^-4 of that as a tolerance for maximum absolute mesh movement between iterations. Then we enter a loop where the maximum absolute movement of any vertex from one iteration to the next is computed until the maximum absolute movement between two iterations is smaller than the tolerance.

# What we changed in the provided code

The provided code instructed us with a comment that we should use the deformed mesh if it exists instead of the source mesh (TODO: Check for an existing deformed mesh, if so use that as an iteration, if not use a mesh named 'source' as the initial mesh). However, when using get_deformed_object to get the deformed_mesh in ARAP, this will result in source and deformed pointing to the same object, which means no new object is made, nor will there be any differences to the existing object So we left it out deemed pointless. We added a timer for in the system console to see how long each part of the code takes.