Introduction to Eigen

Everton Rufino Constantino
Linaro Engineer

Eigen

From the website:

Eigen is a C++ template library for linear algebra: matrices, tensors, vectors, numerical solvers, and related algorithms.

https://eigen.tuxfamily.org/

Eigen is Open Source

Recent versions of the Eigen project (3.1.1+) are licensed under the MPL2 license:

https://www.mozilla.org/en-US/MPL/2.0/

Eigen is flexible and reliable

Choice in data types

- Floating-point
- Integer
- Complex
- User-defined

Choice in matrix types

- Storage: Dense & sparse
- Size: Fixed & dynamic

Choice in algorithms

- Decompositions

Flexibility in:

- Performance
- Memory usage
- Use Cases

Reliability in:

- Accuracy
- Precision
- Stability

Eigen is cross-platform and fast

Eigen is Standard C++14 code

- Works on any platform with a C++14 compliant compiler.
- Uses compile-time features to improve performance.

Also has tuned vectorized implementations for many Instruction Sets:

- Arm: Neon (AArch32, AArch64), SVE
- X86: SSE 2/3/4, AVX, AV2, FMA, AVX512
- Power: AltiVec/VSX (32- & 64-bit)
- s390/zEC13: ZVector
- GPU: Cuda, AMD/HIP, Sycl

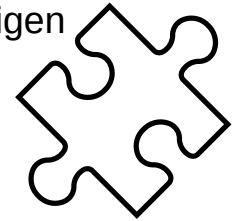
Eigen is a “building block” library

Other libraries build on top of Eigen to provide higher level functionality.

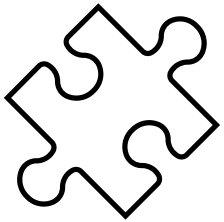
So:

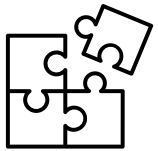
- you may be using it even if you are not aware of it
- It is a foundational library important to the Arm Ecosystem

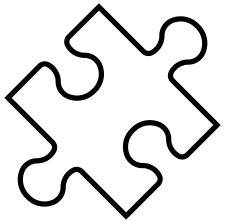
Eigen



GDL - GNU Data Language:
(Visualization)

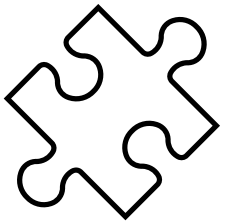






Tensorflow (Machine learning)

ROS – (RoboticOS)



Eigen – How to use it?

https://gitlab.com/everton.constantino/eigen-pagerank-webinar

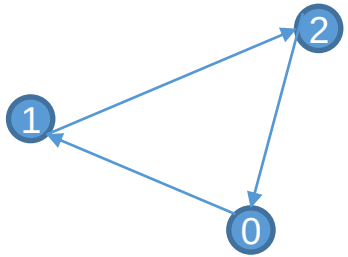
Linaro

Graph/Networks

- Given a set V and a map E: V → V we call the pair G(V,E) a graph.
- Elements of V are called vertices, the relations between elements of V defined by E are called edges.
- The order of G is defined as #V.
- An example: G : (V = {0,1,2}; E = { (0,1), (1,2), (2,0) }).

Graph/Networks

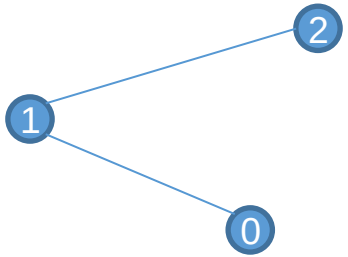
- G can also be represented as the following matrix M
  - $M = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$
    - Where $M_{ij} = \begin{cases} 1 \; if \; (i,j) \in E \\ 0 \; otherwise \end{cases}$



- The representation above is called the adjacency matrix of G.
- G can also be represented graphically.

Graph/Networks

- If E is symmetric G is called an undirected graph, when G is directed (i,j) will be defined as the edge that connects i to j.
- G = (V = {0, 1, 2}; E = { (0,1), (1,0), (1,2), (2,1) })
- With adjacency matrix M



$$M = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

- When G is undirected it's easy to see that M will also be symmetric.
- The degree of a vertex u can be calculated as $d(u) = \sum_{i=0}^{N} M_{ui}$

Dense matrices in Eigen

```
#include <Eigen/Dense>
#include <iostream>

using MyDynamicMatrix = Eigen::Matrix<float, Eigen::Dynamic, Eigen::Dynamic>;

int main(int argc, char *argv[])
{
    MyDynamicMatrix mdm = MyDynamicMatrix::Random(4,4);
    std::cout << mdm << std::endl;
    return 0;
}
```

- Eigen is a header only library, just include the directory.
- We will see how to use CMake later.
- Dense matrices can be either dynamically allocated or have a fixed size.
- Several types of scalar available, even custom ones although not all are guaranteed to be vectorized.
- Several ways to initialize a matrix, here we fill it with random numbers.
- Row and Column vectors are simply normal matrices with 1 row or 1 column.

Dense matrices in Eigen

```
using MyFixedMatrix = Eigen::Matrix<float, 4, 4>;

int main(int argc, char *argv[])
{
    MyFixedMatrix mfm = MyFixedMatrix::Random();
    std::cout << mfm << std::endl;
    return 0;
}
```

Dense matrices in Eigen

```
using namespace Eigen;

void foo()
{
    Matrix2f mf;
    mf << 2,2,2,2;
    MatrixXf Mf(2,2);
    Mf << 1,2,3,4;
}
```

- Predefined fixed and dynamic types.
- Streams for initialization.

Dense matrices in Eigen

```
MatrixXf Mf(2,2);
Mf(0,0) = 0.1f;
Mf(0,1) = 1.23f;
Mf.coeffRef(1,0) = 5.813f;
Mf.coeffRef(1,1) = 21.3455f;
```

- Coefficient access with range check via operator() or without via coeffRef.

Dense matrices in Eigen

```
float *pM = new float[4];                                          M:
pM[0] = 1; pM[1] = 2; pM[2] = 3; pM[3] = 5;                        1 3
Map<MatrixXf> M(pM, 2, 2);                                         2 5
std::cout << "M:" << std::endl << M << std::endl;                  A:
Map<Matrix<float, Dynamic, Dynamic, RowMajor>> A(pM, 2, 2);        1 2
std::cout << "A:" << std::endl << A << std::endl;                  3 5
```

- Already loaded data can be mapped into Eigen objects directly via Map.
- Only one template argument required but you can give more Information to help the mapping.
- Mapped matrices work essentially as normal Eigen ones.

Adding Eigen to a CMake script

```
cmake_minimum_required(VERSION 3.0)
project(eigen_webinar)

find_package(Eigen3 3.4 REQUIRED NO_MODULE)

add_executable(example example.cpp)
target_link_libraries(example Eigen3::Eigen)
```

- Just add Eigen as a target library.
- In case you have it installed somewhere that is not default just set CMAKE_PREFIX_PATH. For example:
  cmake -DCMAKE_PREFIX_PATH=$HOME/sources/eigen ..

Sparse matrices

```
#include <Eigen/Sparse>
#include <iostream>

using namespace Eigen;

int main(int argc, char *argv[])
{
    SparseMatrix<float> M(2,2);
    M.coeffRef(0,0) = 4;
    M.insert(1,0) = 2;
    std::cout << M << std::endl;
    return 0;
}
```

```
Nonzero entries:
(4,0) (2,1) (_,_) (_,_)

Outer pointers:
0 4  $
Inner non zeros:
2 0  $

4 0
2 0
```

- You can add element by element, but you can also add via triplets.

Sparse matrices

```
SparseMatrix<float> M(2,2);
std::vector<Triplet<float>> T;
T.emplace_back(0,0,1);
T.emplace_back(0,1,2);
T.emplace_back(1,0,3);
T.emplace_back(1,1,5);
M.setFromTriplets(T.begin(), T.end());
```

- You can use triples to insert elements more elegantly.
- If you do end up adding elements via insert or coeffReff, make sure to reserve the expected amount of non-zero elements per column. Inserting a new element takes O(n) where n is the number of non-zero elements.

Graph class

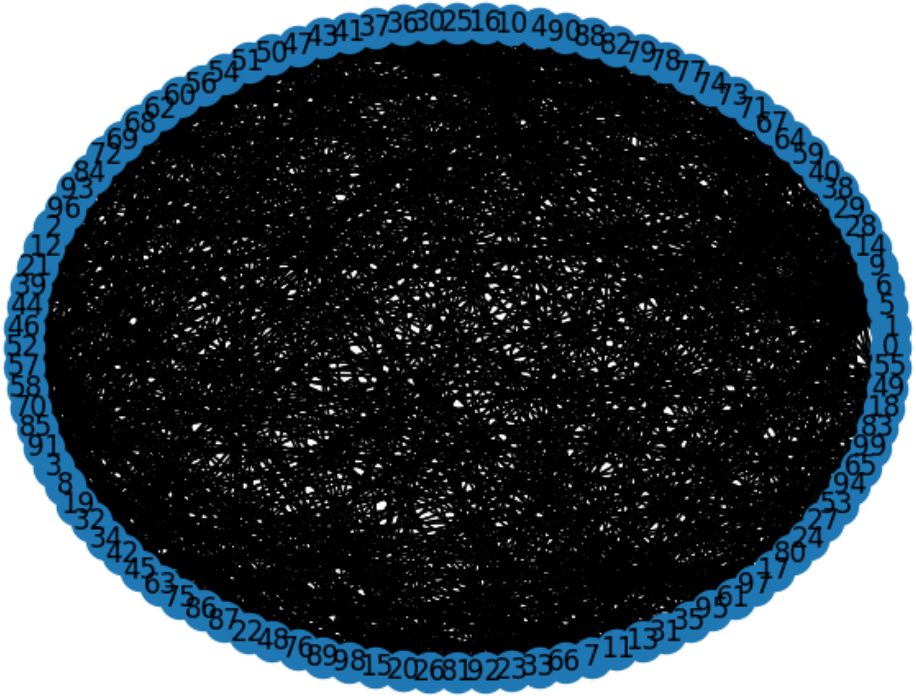
```cpp
using namespace Eigen;

using SMatrix = SparseMatrix<float>;
using DMatrix = Matrix<float, Dynamic, Dynamic>;

template<typename MatrixType, bool isDirected=false>
class Graph
{
    MatrixType adjM_;
    int nodes_;
public:
    void insert_edge(long i, long j)
    {
        if(i == j) return;
        if constexpr (std::is_same<MatrixType, SMatrix>::value)
        {
            adjM_.coeffRef(i,j) = 1;
            if constexpr (!isDirected) adjM_.coeffRef(j,i) = 1;
        } else {
            adjM_(i,j) = 1;
            if constexpr (!isDirected) adjM_(j,i) = 1;
        }
    }

};
```

Random undirected graphs

```
void generate_random_graph(long nodes,
                           double p)
{

    for(auto i = 0; i < nodes; i++)
    {
        for(auto j = i; j < nodes; j++)
        {
            if(dist_(generator_) <= p)
            {
                insert_edge(i,j);
            }
        }
    }
}
```

- Erdös-Rényi model – The naïve random graph. Given a fixed probability p and a number N, G(V,E) is such that #V = N and for all possible $\frac{N(N-1)}{2}$ edges we randomly select them using p. The expected number of edges is then $E(\#E) = \binom{N}{2}p$.
- $p = \frac{\log N}{N}$ is a threshold to graph connectedness.

Random undirected graphs



Random undirected graphs

- Scale free networks
  - On the Erdös-Rényi model the degree distribution can be easily proven to be a binomial

$$\mathrm{P(d(u) = k)} = \binom{N-1}{k} p^k (1-p)^{N-1-k}$$

  - Scale free networks have a degree distribution that are close to $P(d(u) = k) \sim k^{-\alpha}$. They model real world networks much more closely e.g., social networks, webgraphs, collaboration graphs (Erdös number).
- Barabasi-Albert model
  - Start with a random graph $G(V_{m_0}, E_{m_0})$.
  - For each new node added attach it to the previously added ones with probability

$$p_{v_i} = \frac{d(v_i)}{\sum_{k \in V} d(v_k)}$$

Random undirected graphs

Generating a K_n subgraph – Block operations

```
MatrixXf M = MatrixXf::Zero(8,8);
M.topLeftCorner(3,3) = Matrix<float, 3, 3>::Constant(1);
std::cout << M << std::endl;
```

```
1 1 1 0 0 0 0 0
1 1 1 0 0 0 0 0
1 1 1 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

```
MatrixXf M = MatrixXf::Zero(8,8);
M.topLeftCorner(3,3) = Matrix<float, 3, 3>::Constant(1);
M.topLeftCorner(3,3) -= Matrix<float, 3, 3>::Identity();
std::cout << M << std::endl;
```

```
0 1 1 0 0 0 0 0
1 0 1 0 0 0 0 0
1 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

Random undirected graphs

```
MatrixXf M = MatrixXf::Zero(8,8);
const int size = 4;
auto pos = 2;
M.block(pos, pos, size, size) =
   Matrix<float, size, size>::Constant(1);
M.block(pos, pos, size, size) -=
   Matrix<float, size, size>::Identity();
std::cout << M << std::endl;
```

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 1 1 1 0 0
0 0 1 0 1 1 0 0
0 0 1 1 0 1 0 0
0 0 1 1 1 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

- Block operations can be calculated at any position. Here we generate a K_n subgraph starting from any position.
- Notice that block operations can be used either as *rvalues* or *lvalues*.

Random undirected graphs
Summing all degrees – Eigen Reductions

- We then want $D = \sum_{u \in V} d(u)$.

```
MatrixXf M = MatrixXf::Ones(4,4);
std::cout << "Sum: " << M.sum() << std::endl;
std::cout << "Prod: " << M.prod() << std::endl;
std::cout << "Mean: " << M.mean() << std::endl;
std::cout << "Trace: " << M.trace() << std::endl;
```

```
Sum: 16
Prod: 1
Mean: 1
Trace: 4
```

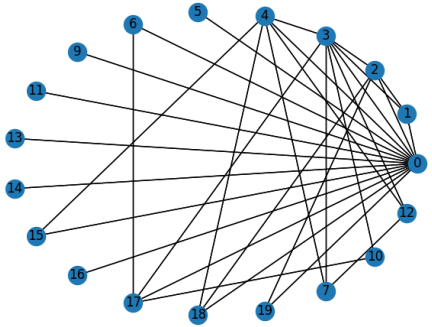
- Fast calculation over all coefficients.
- To get the degree of a particular vertex we can just sum over its column: `adjM_.col(u).sum()`
- How to do this for sparse matrices?

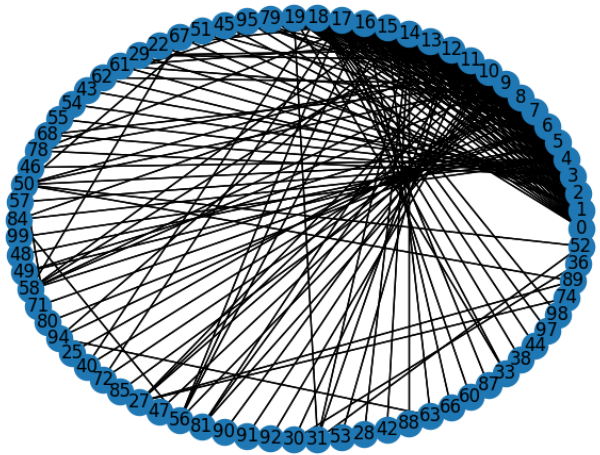
Random undirected graphs
Summing all degrees – Iterating over sparse matrices

```
for(auto i = 0; i < adjM_.outerSize(); i++)
{
    for(SMatrix::InnerIterator it(adjM_, node); it; ++it)
        sumDeg++;
}
```

Random undirected graphs
Barabasi-Albert

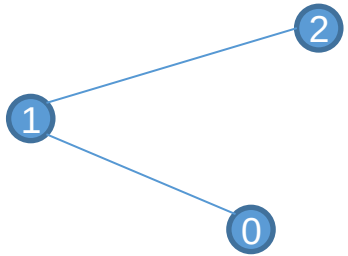
```
void generate_random_graph()
{

    long m0 = nodes_*0.01 >= 2 ? nodes_*0.01 : 2;
    long sum = 0;

    /* Start with K_m0 */

    sum = sum_degree();

    for(auto i = m0; i < nodes_; i++)
    {
        for(auto j = 0; j < i; j++)
        {
            auto k = degree(j);
            auto prob = k/static_cast<double>(sum)

            if(dist_(generator_) <= prob)
            {
                insert_edge(i,j);
                sum += 2;
            }
        }
    }
}
```





Walks, paths and trails

- A walk is defined as a sequence of edges $S = (s_1, s_2, \dots, s_n)$, finite or not, the set of vertices spanned by S we will call $S_v = (v_1, v_2, \dots, v_n)$ then we must have $W(s_i) = (v_i, v_{i+1})$. A trail is defined as a walk with no repeating edges and a path as a trail with no repeating vertices.
- On G a possible walk would be $W = \{(0,1), (1,2), (2,1), (1,0), (0,1)\}$, a trail $T = \{(0,1), (1,2)\}$ notice that on this particular graph all trails are paths.
- A circuit is a path that starts and ends on the same vertex.



Powers of the adjacency matrix

- Let M be the adjacency matrix of G then $M_{ij}^2 = \sum_{k \in V} M_{ik} M_{kj}$, it's easy to see that every time there is a walk of length 2 from i to j then that walk will be counted on M^2. What is M_{ii}^2?
- What happens if we go to the next power of M?

$$M_{ij}^3 = \sum_{z \in V} M_{iz}^2 M_{zj} = \sum_{z \in V} (\sum_{k \in V} M_{ik} M_{kz}) M_{zj}$$

The innermost summation counts the number of walks of length 2 from i to z and the outermost one counts the number of walks from i to j of length 3.

Walks on G – Powers of the adjacency matrix

```
#include <unsupported/Eigen/MatrixFunctions>
/*...*/
bool pathFromTo(long u, long v, long pathLength)
{
    if constexpr (std::is_same<MatrixType, SMatrix>::value)
    {
        MatrixType P(adjM_);
        for(auto i = 1; i < pathLength; i++)
            P = P*P;
        return P.coeff(u,v) > 0;
    } else {
        MatrixType P = adjM_.pow(pathLength);
        return P.coeff(u,v) > 0;
    }
    return false;
}
```

- Unsupported is not as unsupported as it sounds.
- Alias? What is aliasing?

WARNING

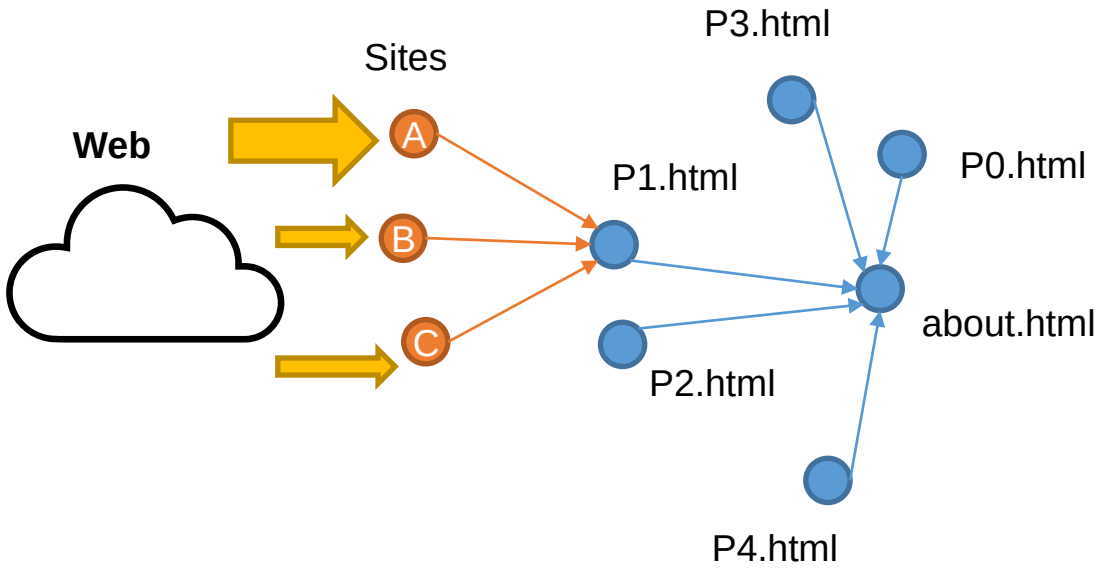
- Refer to http://eigen.tuxfamily.org/dox-devel/TopicPitfalls.html
- Aliasing – you can use **eval()** to solve this but be careful with it.
- Eigen has a complex expression solver forcing evaluation gives the compiler much less freedom to optimize so use it wisely.
- **auto** keyword – Don’t use it because of the reason above.
- Pass by value – Just don’t.

Centrality measures

- How do you measure the relevance of a particular vertex?
- First order measure could be the degree of that vertex, however that doesn't translate some properties appropriately. For example

- Centrality measures try to solve the problem of capturing “important” nodes.
- We can count the degree but add more weight if that vertex is connected to more relevant ones. Let C(v) be the centrality measure then define

$$C(v) = \alpha \sum_{t \in V} C(t) M_{vt}$$



Centrality measures – Eigenvenctor centrality

```
template<typename ComplexScalar, typename EigenVectorType>
ComplexScalar eigenvector_centrality(EigenVectorType& v)
{
    EigenSolver<MatrixType> solver(adjM_);

    typename MatrixType::Index mIdx;
    solver.eigenvalues().real().maxCoeff(&mIdx);

    v = solver.eigenvectors().col(mIdx);

    return solver.eigenvalues()[mIdx];
}
```

- We are applying this function only for undirected graphs, so the adjacency matrix is symmetric and by the Spectral theorem only possesses real eigenvalues.

Centrality measures – Eigenvector centrality

```
Graph<DMatrix> G(sz);
typename EigenSolver<DMatrix>::EigenvectorsType v;
G.eigenvector_centrality<EigenSolver<DMatrix>::ComplexScalar,
EigenSolver<DMatrix>::EigenvectorsType>(v);
```

- Eigen provides information on the expected Eigenvector and Eigenvalue types that can be easily extracted.
- Currently, Eigen does **not** provide the Eigensolver functionality on Sparse matrices.

Pagerank

- The Pagerank algorithm tries to calculate the probability that randomly clicking on pages will make you arrive at another.
- It's another sort of centrality measure where PR is actually a probability distribution.
- It adds a dumping factor, which translates as just stopping the random clicks.
- Also adds a bias, people could have the link from outside the web for example.
- This is just a stochastic variant of the Eigenvector centrality measure.
- First transform the adjacency matrix into a stochastic matrix, where each column has norm 1.
- Then for each step $\mathrm{pr}_{t+1} = \widehat{M} pr_t$ where $\widehat{M} = \frac{1-d}{N} + dMD^{-1}$ and D is just a diagonal matrix with the norm of each column as its entry making MD^{-1} a stochastic matrix.
- Either iterate for a fixed number of times or when $|pr_{t+1} - pr_t| < \epsilon$.

Pagerank
Calculating $\widehat{M}$ -Partial reductions, inverses and beyond

```
MatrixType D = adjM_h.colwise().sum().asDiagonal();

adjM_h = d*adjM_h*D.inverse();
adjM_h.array() += (1-d)/nodes_;
```

- Use **colwise()** or **rowwise()** to apply operations to each column or row and return it as a vector. Here **colwise().sum()** returns a vector with each element being the sum of each column.
- **asDiagonal()** apply only to vectors and construct a square matrix where the diagonal has the same elements as the vectors.
- Finding the inverse of a matrix, when it exists is quite easy with Eigen. If you need to check for inversibility refer to **MatrixBase::inverse** documentation on https://eigen.tuxfamily.org/dox/. Notice, using the inverse here is a bad idea and just for the sake of demonstration.
- **array()** 'converts' the matrix into an array for coefficient-wise operations, without additional computational costs.

Pagerank – Power method

```
VectorXf v0 = v;

v = adjM_h*v0;
for(auto i = 0; i < iters; i++)
{
    v0 = v;
    v = adjM_h*v0;
}

return (v - v0).norm();
```

- Vectors also have predefined types like a dynamically allocated float vector through **VectorXf.**
- Use **norm()** to get the squared root of the usual Euclidian norm which can be calculated via **squaredNorm().**
- **norm()** also work on matrices and return the Frobenius norm.
- Other ℓ^p norms can easily be calculated using **lpNorm<p>()**

Closing remarks

- http://eigen.tuxfamily.org/dox/AsciiQuickReference.txt
- Eigen does not rely on auto vectorization.
- Several other modules
  - Tensors with algebra.
  - Geometry with transformations, rotations, quarternions…
  - Fast vectorized special functions like log, trigonometric functions, gamma, logistic…
  - Deep linear algebra functionality with several solvers, determinant, least squares…
  - Much more decompositions like LU, QR, LLT, LDLT, SVD, Schur…

Eigen future

- Faster GEMM, with mixed precision calculation and easier customization.
- Parallelization on dense matrices.
- Tensor on Core.
- Threading with custom ThreadPools.
- Support for bfloat16 on Arm.

Thank you

- Do you need help with Eigen in particular or more general performance optimization, toolchain enablement, HPC or Machine Learning? Come talk to us!
- softwareservices@linaro.org

Linaro