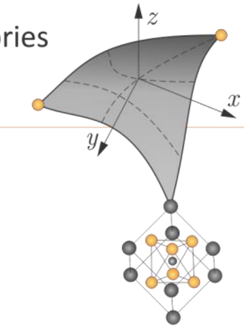


A Case Study on Design and Performance of a Generic Finite Element Library

Yingrui (Ray) Chang

Graduate Aerospace Laboratories
California Institute of Technology

Congress on Strange Mechanics
Weird Place, CA
February 31, 2013

Education & Experience

Yingrui (Ray) Chang
California Institute of Technology

Education:

- **Doctor of Philosophy (Ph.D.)** in Mechanical Engineering, 2010 - present
  Minor in CSE (Computational Science and Engineering)
  California Institute of Technology, CA.
- **Master of Science (M.S.)** in Computational Mechanics, 2009 - 2010
  Carnegie Mellon University, PA .
- **Bachelor of Engineering (B.E.)** in Civil Engineering, 2005 - 2009
  Tongji University, Shanghai, China.

Experience:

- **model building and solving:**
  - building and solving partial differential equations (**PDEs**) arising in mechanics and material science;
  - numerical PDEs, linear algebra, optimization, etc.
- **programming skills:**
  - building C++ numerical finite element (FEM) libraries for solving PDEs;
  - high-performance computing (MPI, openmp);
  - other languages: python, Java, C.

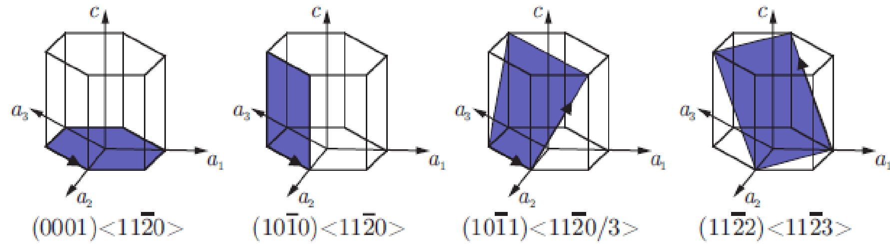
Magnesium and Magnesium Alloys

Yingrui (Ray) Chang
California Institute of Technology

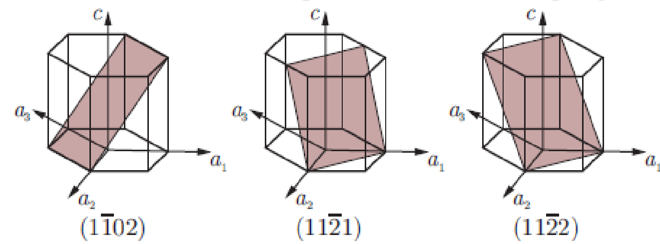
general properties:

atomic number: 12
melting point: 923 K (650 ˚C)
density: 1.738 g/cm^3
(Fe: $7.8 g/cm^3$; Al: 2.7 g/cm^3)
Young's modulus: 45 GPa
shear modulus: 17 GPa
crystal structure: hcp

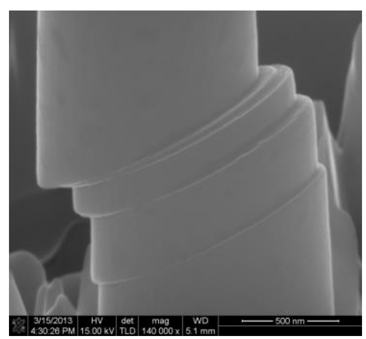
crystal plasticity: hcp slip systems



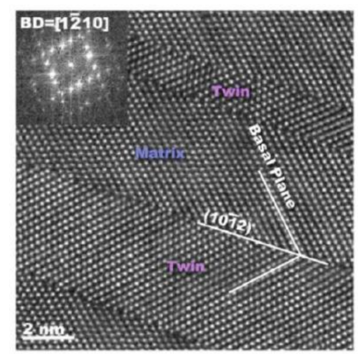
deformation twinning: hcp twinning systems



experimental evidence of twinning and slip in magnesium:



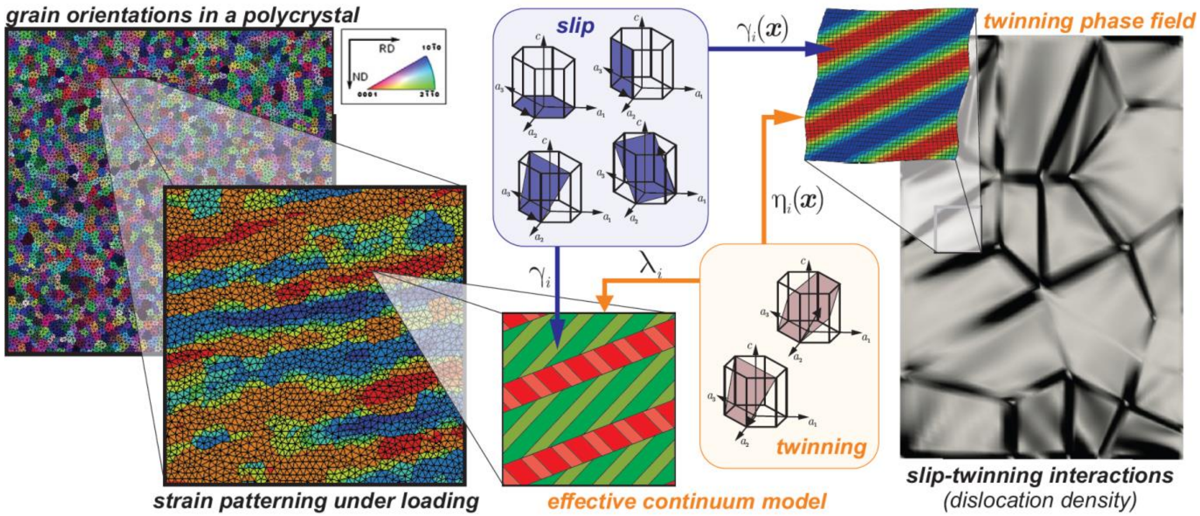
(Z. Aitken, J. Greer, 2014)



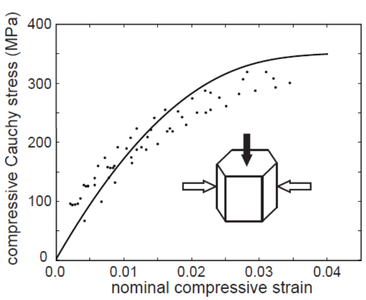
(Yu, Qi, Chen, Mishra, Li &Minor. 2011)

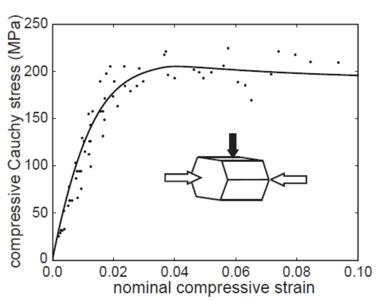
Building Models for Magnesium and Mg Alloys

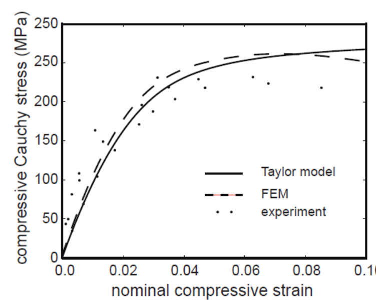
Yingrui (Ray) Chang
California Institute of Technology

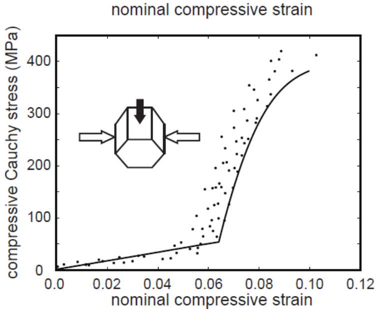


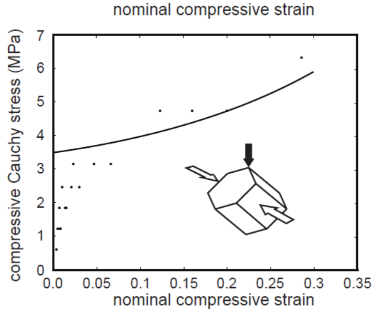
grain structure and FE mesh representation:



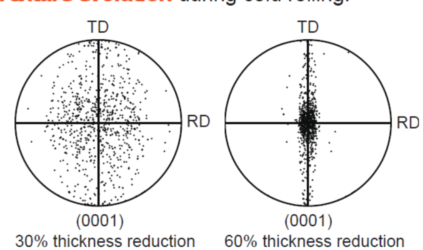








Texture evolution during cold rolling:



microstructure formation:

unpublished result here.

Presentation Outline

Yingrui (Ray) Chang
California Institute of Technology

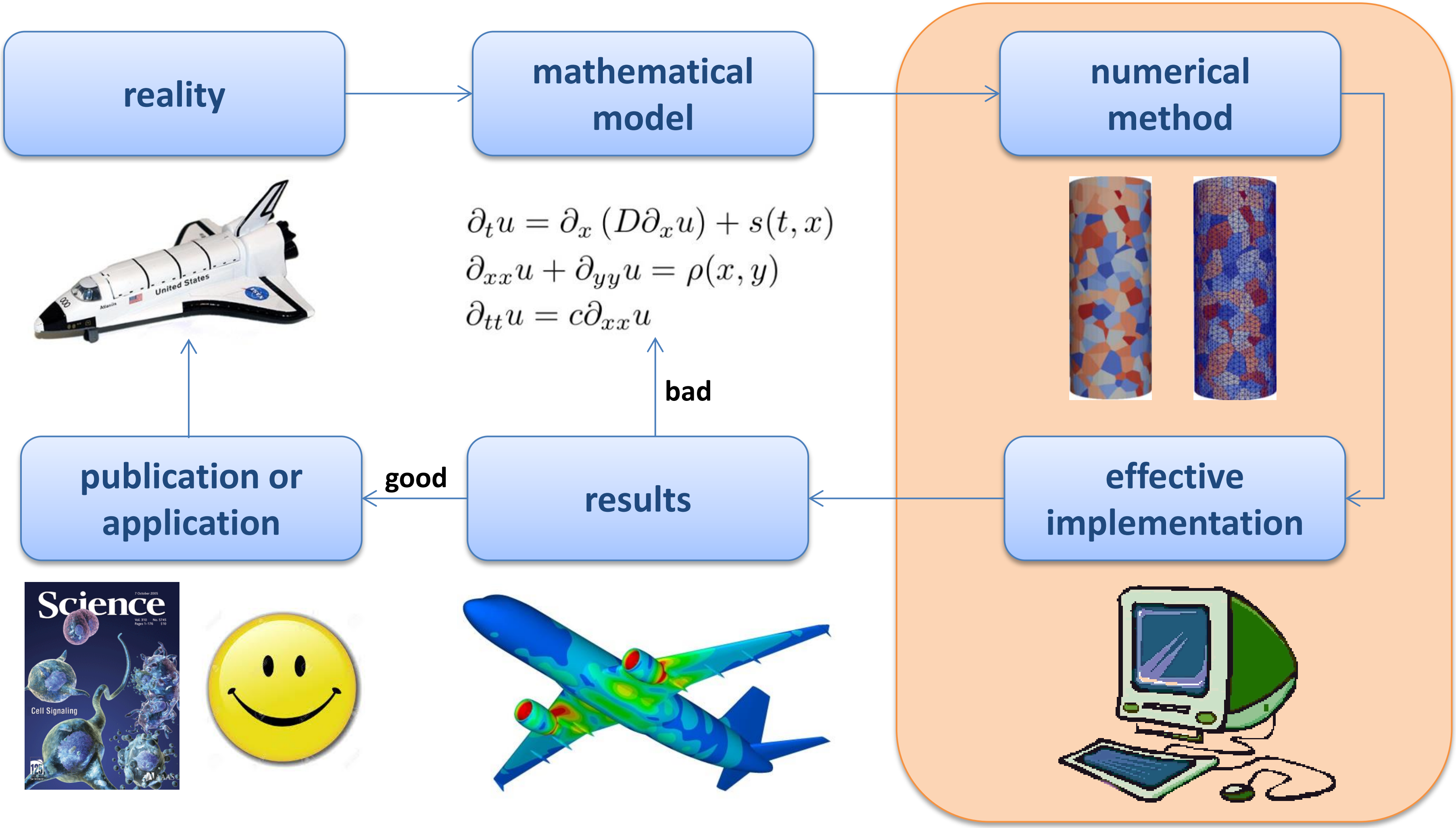
- **Brief introduction to finite element method (FEM):**
  - what problem we are solving;
  - FEM domain discretization;
  - role of **assembler class**;
- **Design study for building generic FEM code:**
  - inheritance (virtual function) approach;
  - template approach;
- **Performance study on threading FEM assembler class:**
  - assemble energy;
  - assemble force vector;
  - assemble tangent matrix.

Typical Model Building Flow Chart

Yingrui (Ray) Chang
California Institute of Technology

Computational Models in Engineering

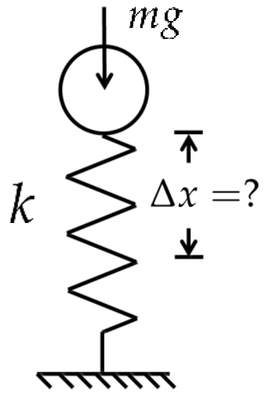
How we do what we do…



Energy Minimization

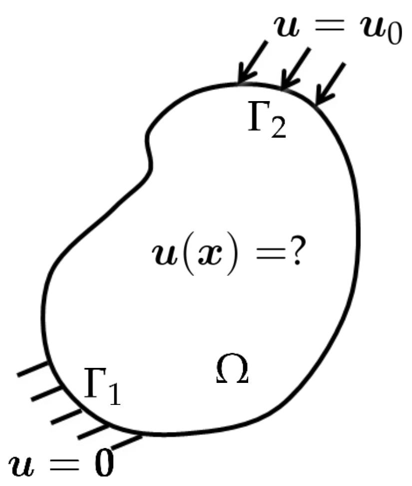
Yingrui (Ray) Chang
California Institute of Technology

energy minimization of single spring:



- system energy : $\Psi(\Delta x) = -mg\Delta x + \frac{1}{2}k\Delta x^2$
- minimize Ψ : $\frac{\mathrm{d}\Psi}{\mathrm{d}\Delta x} = 0 \quad \Rightarrow \Delta x = \frac{mg}{k}$

solid mechanics example:

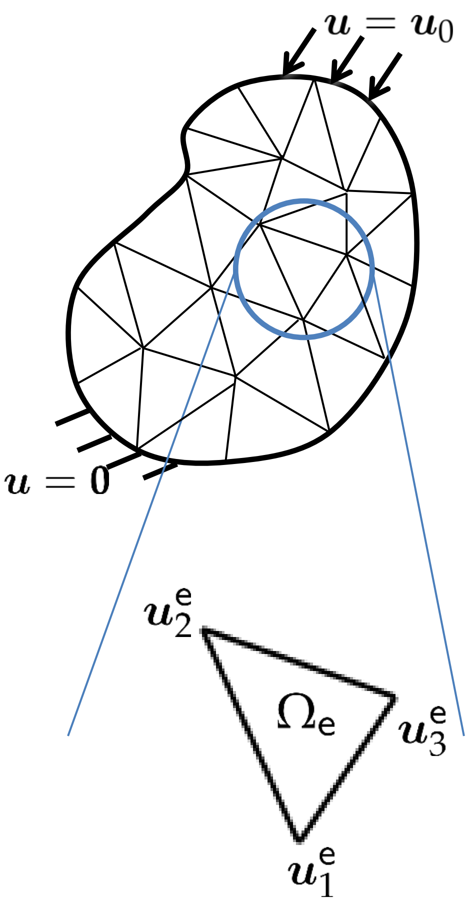


- unknown: displacement field: $\boldsymbol{u}(\boldsymbol{x})$
- local stored energy : $W(\nabla \boldsymbol{u}(\boldsymbol{x}))$
- system energy : $\Psi(\boldsymbol{u}(\boldsymbol{x})) = \int_\Omega W(\nabla \boldsymbol{u}(\boldsymbol{x}))\mathrm{d}\,\boldsymbol{v}$
- **our problem:**
  $\boldsymbol{u}(\boldsymbol{x}) = \arg\min \Psi(\boldsymbol{u}(\boldsymbol{x}))$
  subject to: $\boldsymbol{u}(\boldsymbol{x}) = \boldsymbol{0}$ on Γ_1, $\quad \boldsymbol{u}(\boldsymbol{x}) = \boldsymbol{u}_0$ on Γ_2

Finite Element Method

Yingrui (Ray) Chang
California Institute of Technology

finite element discretization:



FEM idea:

- discretize the domain in space using elements;
- use unknowns on the nodes to approximate the unknown field;
- transform the original variational problem to a **discretized optimization problem**.

- displacement discretization : $\boldsymbol{u}(\boldsymbol{x}) \approx \boldsymbol{U} = [\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots \boldsymbol{u}_n]^{\mathrm{T}}$
- system energy is the summation from all elements :

$$\Psi(\boldsymbol{u}(\boldsymbol{x})) = \Psi(\boldsymbol{U}) = \sum \int_{\Omega_e} W(\boldsymbol{U}_e) \mathrm{d}\,\boldsymbol{v}$$

- stationary point, force vector : $\boxed{\boldsymbol{F}(\boldsymbol{U}) = \frac{\partial \Psi}{\partial \boldsymbol{U}} = \boldsymbol{0}}$ subject to B.Cs.

system of nonlinear equations.

Finite Element Method

Yingrui (Ray) Chang
California Institute of Technology

our problem:

find displacements at each node $\boldsymbol{U}_s = [\boldsymbol{u}_1, \boldsymbol{u}_2, ...\boldsymbol{u}_n]^{\mathrm{T}}$ satisfying $\boldsymbol{F}(\boldsymbol{U}_s) = \frac{\partial \Psi}{\partial \boldsymbol{U}} = \boldsymbol{0}$

numerical root finding via Newton's tangent method:

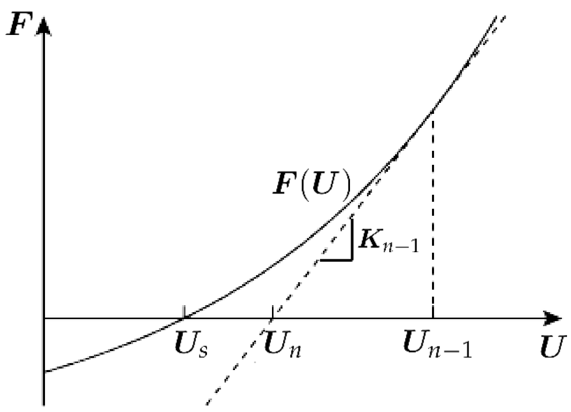
- given initial guess: $\boldsymbol{U}_0$
- do the following updates until $\boldsymbol{F}(\boldsymbol{U}_n) \approx \boldsymbol{0}$

$$\boldsymbol{U}_n = \boldsymbol{U}_{n-1} - \boldsymbol{K}_{n-1}^{-1} \boldsymbol{F}(\boldsymbol{U}_{n-1}), \quad \text{where} \quad \boldsymbol{K} = \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{U}} = \frac{\partial^2 \Psi}{\partial \boldsymbol{U} \partial \boldsymbol{U}}$$

given displacements at each node $\boldsymbol{U}$, we need to compute:

- system energy: $\Psi(\boldsymbol{U})$
- system force vector: $\boldsymbol{F}(\boldsymbol{U}) = \frac{\partial \Psi}{\partial \boldsymbol{U}}$
- system tangent matrix: $\boldsymbol{K}(\boldsymbol{U}) = \frac{\partial^2 \Psi}{\partial \boldsymbol{U} \partial \boldsymbol{U}}$

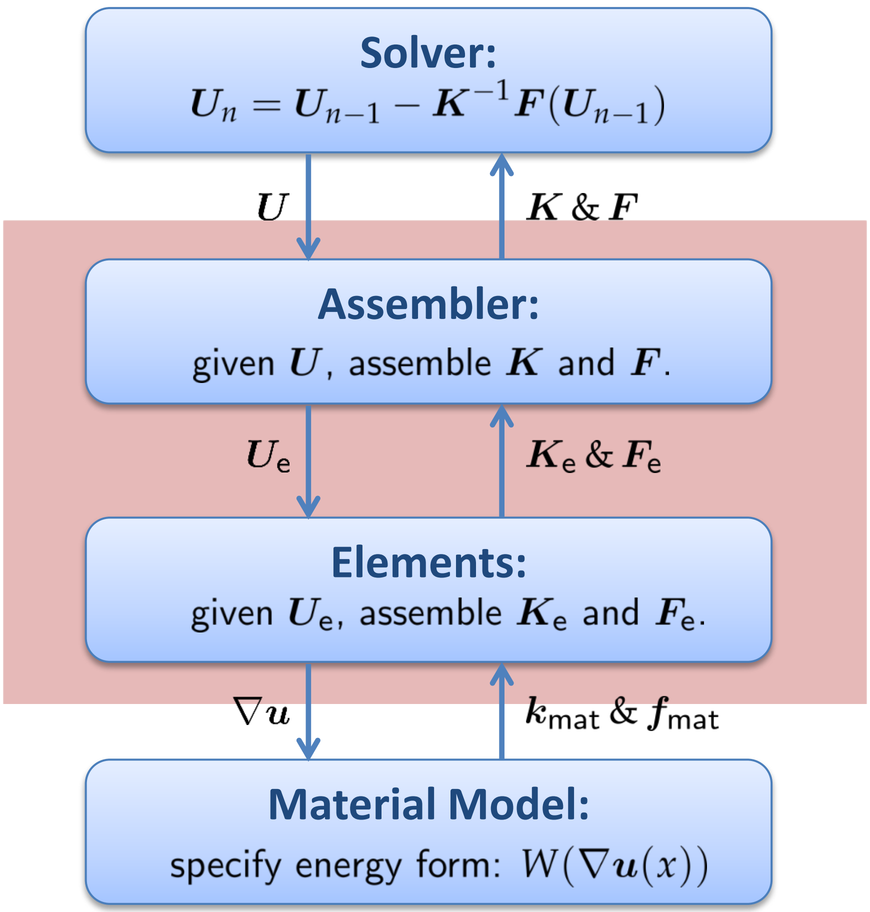
Newton's method illustration:



General Finite Element Code Building Structure

Yingrui (Ray) Chang
California Institute of Technology

generic finite element code structure:



example:
Newton's method, conjugate gradient, …

example:
triangle, quadrilateral, tetrahedron,. …

example:
elastic, plastic, viscosity, …

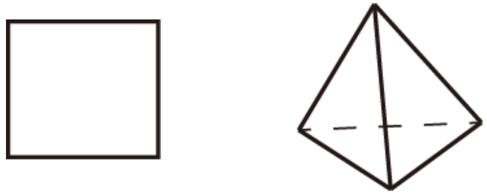
Element & Assembler

Yingrui (Ray) Chang
California Institute of Technology

Element class member functions:

```
elementEnergy computeEnergy(dispsAtElementNodes);
elementForces computeForce(dispsAtElementNodes);
elementTangentMatrix computeTangentMatrix(dispsAtElementNodes);
```

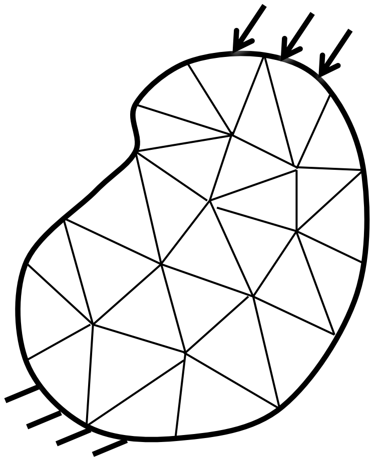
quad and tet elements:



Assembler class member functions:

```
systemEnergy assembleEnergy(dispsAtGlobalNode){
  double energy=0;
  for (eleId=0; ...){
    //get displacements of current element
    energy+=elements[eleId].computeEnergy(eleDisps);
  }
  return energy;
};
systemForces assembleForceVector(dispsAtGlobalNode) {...};
systemTangentMatrix assembleTangentMatrix(dispsAtGlobalNode){...};
```

global system:



Design By Inheritance and Virtual Functions

Yingrui (Ray) Chang
California Institute of Technology

VectorXd && MatrixXd:

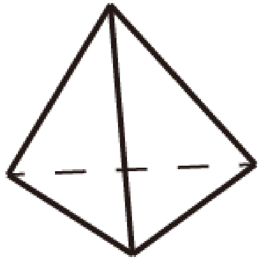
dynamical allocated vector/matrix the size of which is determined at runtime.

element interface:

```
class ElementBase {
public:
  using ElementVector = VectorXd; //used to expression the dimensionality
  using ElementDisplacements = std::vector<ElementVector>; //size of vector = numberOfNodes
  using ElementForce = std::vector<ElementVector>;
  using ElementTangentMatrix = MatrixXd;

  virtual double computeEnergy(const ElementDisplacements&) const = 0;
  virtual ElementForce computeForce(const ElementDisplacements&) const = 0;
  virtual ElementTangentMatrix computeTangentMatrix(const ElementDisplacements&) const = 0;
}
```

tetrahedron element:



specific element implementation:

```
class Tetrahedron : public ElementBase {
public:
  double
  computeEnergy(const ElementBase::ElementDisplacements&) const override {...};

  ElementBase::ElementForce
  computeForce(const ElementBase::ElementDisplacements&) const override {...};

  ElementBase::ElementTangentMatrix
  computeTangentMatrix(const ElementBase::ElementDisplacements&) const override {...};
}
```

Design By Inheritance and Virtual Functions

Yingrui (Ray) Chang
California Institute of Technology

assembler class should use pointers to the interface

```
class Assembler {
public:
  using Vector = VectorXd;
  using Displacements = std::vector<Vector>;
  using ForceVector = VectorXd;
  using TangentMatrix = SparseMatrix;

  double assembleEnergy(const Displacements& globalDisplacements) const ;
  ForceVector assembleForceVector(const Displacements& globalDisplacements) const;
  TangenetMatrix assembleTangentMatrix(const Displacements& globalDisplacements) const;

private:
  std::vector<ElementBase*> elements;
}
```

implementation of energy assembler:

```
double
Assembler::assembleEnergy(const Displacements& globalDisplacements) const {
  double energy=0;
  for (auto& element : elements){
    //get displacement for each element (eleDisps)
    energy+=element->computeEnergy(eleDisps);
  }
  return energy;
};
```

Difficulties with Interface

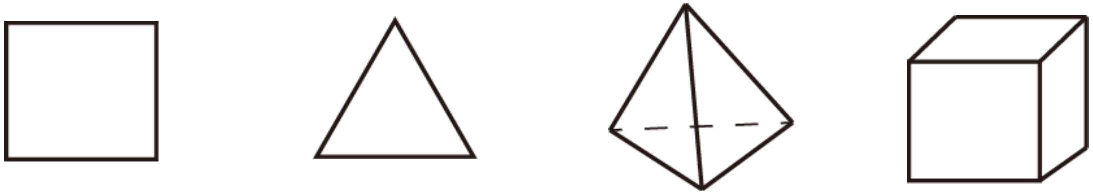
Yingrui (Ray) Chang
California Institute of Technology

element interface:

```
class ElementBase {
public:
  using ElementVector = VectorXd;
  using ElementDisplacements = std::vector<ElementVector>;
  using ElementForce = std::vector<ElementVector>;
  using ElementTangentMatrix = MatrixXd;

  virtual double computeEnergy(const ElementDisplacements&) const =0;
  virtual ElementForce computeForce(const ElementDisplacements&) const =0;
  virtual ElementTangentMatrix computeTangentMatrix(const ElementDisplacements&) const =0;
}
```

different elements have different properties:



problems:

- interface cannot express information about specific properties of each element type (e.g. number of nodes, dimension, etc.)
- checking preconditions incurs heavy run time cost.

Implementation and Performance

Yingrui (Ray) Chang
California Institute of Technology

assembler class containing pointers leads to …

- **choice of pointer:** `std::unique_ptr`, `std::shared_ptr`, raw pointer?
- **more implementation and maintaining work:**
  - copy constructor: `Assembler(const Assembler&);`
  - copy assignment operator: `Assembler& operator=(const Assembler&);`
  - move constructor: `Assembler(Assembler&&) noexcept;`
  - move assignment operator: `Assembler& operator=(Assembler&&) noexcept;`
  - destructor: `~Assembler();`
  - virtual constructor: `Tetrahedron* clone();`

performance problem:

- an extra v-pointer in every inherited class (large data type: typically 8 bytes);
- every function call involves pointer tracing (cache unfriendly);
- harmful to data alignment;
- impossible for function inlining.

(a performance study available at *https://github.com/yingryic/performance_study/*)

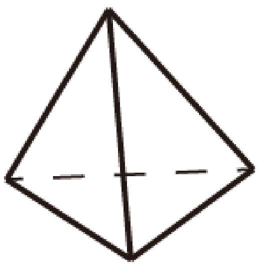
Template Alternative (our approach)

Yingrui (Ray) Chang
California Institute of Technology

`Matrix<double, NumberOfRows, NummberOfCols>`:

statically allocated matrix whose size should be known at compile time.

tetrahedron element:



element classes are not derived from any class:

```
class Tetrahedron {
public:
  const static int NumberOfNodes = 4;
  const static int SpatialDimension = 3;
  const static int NumberOfDofs = NumberOfNodes*SpatialDimension;

  using ElementVector = Matrix<double, SpatialDimension, 1>;
  using ElementDisplacements = std::array<ElementVector, NumberOfNodes>;
  using ElementForce = std::array<ElementVector, NumberOfNodes>;
  using ElementTangentMatrix = Matrix<double, NumberOfDofs, NumberOfDofs>;

public:
  double
  computeEnergy(const ElementDisplacements& elementDispls) const {...};
  ElementForce
  computeForce(const ElementDisplacements& elementDispls) const {...};
  ElementTangentMatrix
  computeTangentMatrix(const ElementDisplacements& elementDispls) const {...};
};
```

Template Alternative (our approach)

Yingrui (Ray) Chang
California Institute of Technology

assembler class contains element objects rather than pointers:

```
template<class ElementType>
class Assembler {
public:
  static const int SpatialDimension = ElementType::SpatialDimension;
  using ElementVector = typename ElementType::ElementVector;
  using ElementDisplacements = typename ElementType::ElementDisplacements;
  using Displacements = std::vector<ElementVector>;
  using ForceVector = VectorXd;

  double assembleEnergy(const Displacements& displs) const {
    double energy=0;
    for (auto& element : elements){
      ElementDisplacement eleDisps=...;
      energy += element.computeEnergy(eleDisps);
    }
    return energy;
  };

  ForceVector assemblerForceVector(const Displacements& displs) const {...};
  SparseMatrix assembleTangentMatrix(const Displacements&displs) const {...};

private:
  std::vector<ElementType> elements;
  ... ...
};
```

Template vs. Virtual Functions

Yingrui (Ray) Chang
California Institute of Technology

advantage of using template:

- dimensionality mismatch can be caught at compile time.
- no extra v-pointer and virtual function calls.
- can rely on the default copy/move constructor/assignment operator, destructor, etc.
- possible for function inlining (compiler can see the implementation).

advice from *Bjarne Stroustrup*:

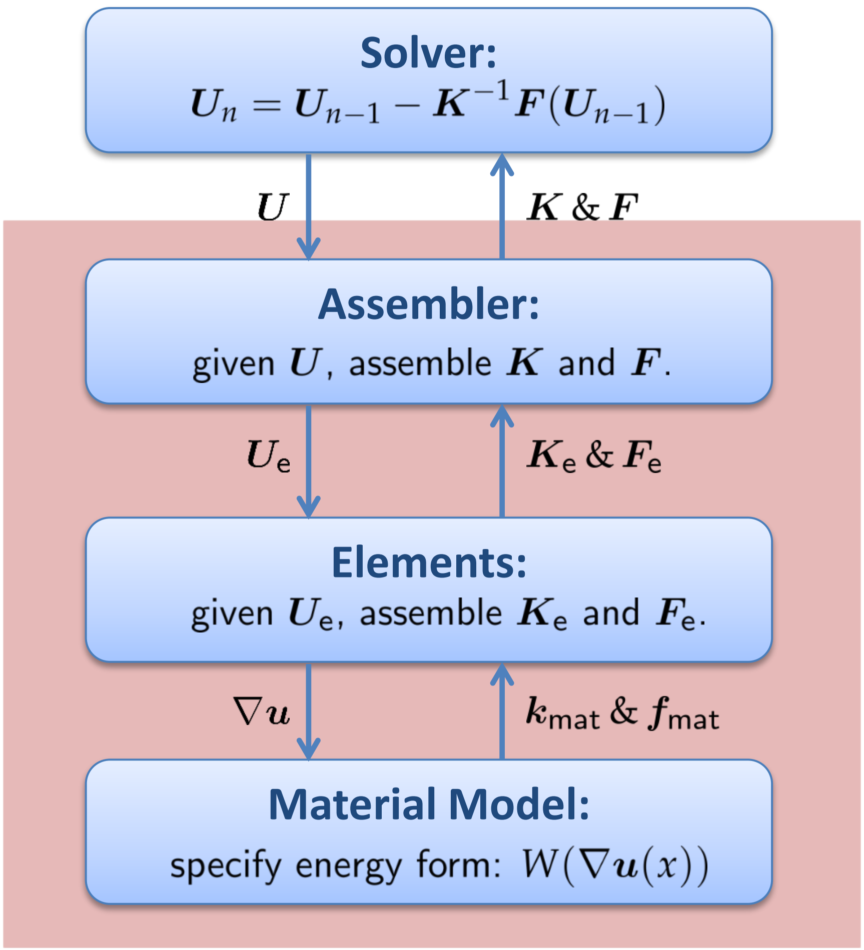
- Prefer a template over derived classes when run-time efficiency is at a premium.
- Prefer derived classes over a template if adding new variants without recompilation is important.
- Prefer a template over derived classes when no common base can be defined.
- Prefer a template over derived classes when built-in types and structures with compatibility constraints are important.

The C++ Programming Language, 3rd Edition, chapter 13.8

Performance Concerns

Yingrui (Ray) Chang
California Institute of Technology

generic finite element code structure:



computation bottle neck:

- **solver level**: time complexity O(n^3); finding solution to system of linear equations
- **assembler level**: time complexity O(n); usually involves complex underlying material models

concurrency techniques:

- **solver level**:
- utilize threaded linear algebra libraries (ViennaCL, Intel Math Kernal Library)
- **threaded assembler.**
- assembleEnergy;
- assembleForceVector;
- assembleTangentMatrix;

Case Study of Threaded Assembler

Yingrui (Ray) Chang
California Institute of Technology

experimental setup:

- parallelize assembly of 681,942 tetrahedron elements in a pillar compression example
- time performance with different number of threads

hardware and system configuration:

- cpu: Intel Xeon E5-2680 CPU, 8 cores;
- memory: 24GB;
- system: Redhat 64 bit;
- compile: g++-4.8.

threading library: openmp

finite element discretization of pillar compression test:

Assemble Energy

Yingrui (Ray) Chang
California Institute of Technology

single thread Energy Assembler (pseudo code):

```
initialize globalEnergy=0

#pragma omp parallel for reduction (+:globalEnergy)
for each element:
  get element displacements and node Id
  compute elementEnergy
  add elementEnergy to globalEnergy

return globalEnergy
```

multithread Energy Assembler (pseudo code):

```
initialize globalEnergy=0
#pragma omp parallel {
  initialize threadLocalEnergy=0
#pragma omp for
  for each element {
    get element displacements and node Id
    compute elementEnergy
    add elementEnergy to threadLocalEnergy
  }
#pragma omp critical
  globalEnergy+=threadLocalEnergy
}
return globalEnergy
```

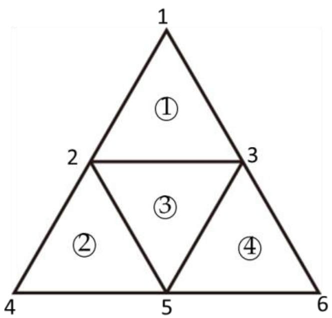
run time of assemble energy with different numbers of threads:

#. of threads	run time (s)	speedup
1	0.1588	1.00
2	0.0805	1.97
3	0.0561	2.83
4	0.0474	3.35
5	0.0378	4.20
6	0.0324	4.90
7	0.0285	5.56
8	0.0251	6.33

Assemble Force Vector

Yingrui (Ray) Chang
California Institute of Technology

system with four elements:



problem dimension: 2;
degree of freedom: 1

$$\boldsymbol{U} : 6 \times 1$$

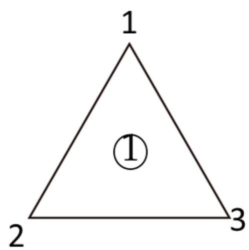
$$\boldsymbol{F} : 6 \times 1$$

$$\boldsymbol{K} : 6 \times 6$$

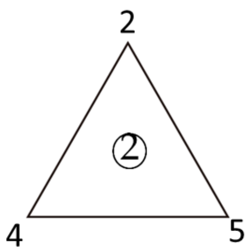
final force vector:

$$\boldsymbol{F} = \begin{bmatrix} F_1^1 \\ F_2^1 + F_1^2 + F_3^3 \\ F_3^1 + F_2^3 + F_1^4 \\ F_2^2 \\ F_3^2 + F_1^3 + F_2^4 \\ F_6^4 \end{bmatrix}$$

assemble force in action:



$$\boldsymbol{F}_{\mathrm{e}}^1 = \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} F_1^1 & F_2^1 & F_3^1 \end{bmatrix} \end{array} \xrightarrow{\text{assemble}} \boldsymbol{F} = \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{bmatrix} F_1^1 & F_2^1 & F_3^1 & . & . & . \end{bmatrix} \end{array}$$



$$\boldsymbol{F}_{\mathrm{e}}^2 = \begin{array}{c} \begin{array}{ccc} 2 & 4 & 5 \end{array} \\ \begin{bmatrix} F_1^2 & F_2^2 & F_3^2 \end{bmatrix} \end{array} \xrightarrow{\text{assemble}} \boldsymbol{F} = \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{bmatrix} F_1^1 & F_2^1 + F_1^2 & F_3^1 & F_2^2 & F_3^2 & . \end{bmatrix} \end{array}$$

source of data race: nodes shared by different elements will be modified by different elements.

Assemble Force Vector (Solution #1)

Yingrui (Ray) Chang
California Institute of Technology

single thread Force Assembler (pseudo code):

```
initialize globalForceVector=0

for each element:
  get element displacements and node Id
  compute elementForce
  add elementForce to globalForceVector

return globalForceVector
```

multithread Force Assembler (pseudo code):

```
initialize globalForceVector=0
#pragma omp parallel {
  initialize a threadLocalForceVector=0
#pragma omp for
  for each element {
    get element displacements and node Id
    compute elementForce
    add elementForce to threadLocalForceVector
  }
#pragma omp critical
  globalForceVector+=threadLocalForceVector
}
return globalForceVector
```

serialized adding vectors may harm performance.

run time of assemble force vector with different numbers of threads:

#. of threads	run time (s)	speedup
1	0.2598	1.00
2	0.1350	1.93
3	0.1033	2.52
4	0.0785	3.32
5	0.0670	3.89
6	0.0569	4.57
7	0.0515	5.05
8	0.0462	5.63

Assemble Force Vector (Solution #2)

Yingrui (Ray) Chang
California Institute of Technology

key idea: each thread writes to the designated position directly.

avoiding data race:

- prepare a number of locks, each lock protects certain nodes;
- when adding entries: lock the corresponding lock, update entry, then release the lock.

multithread Force Assembler (lock version):

```
initialize globalForceVector=0
parallel prepare n locks
#pragma omp parallel for {
  for each element {
    get element displacements and node Id
    compute elementForce
    for each node
      calculate lock id
      set corresponding lock
      add elementForce to globalForceVector
      unset lock
  }
}
return globalForceVector
```

each lock protects 10 nodes:

#. of threads	run time (s)	speedup
1	0.3065	1.00
2	0.1573	1.95
4	0.0979	3.13
6	0.0608	5.05
8	0.0452	6.79

each lock protects 1000 nodes:

#. of threads	run time (s)	speedup
1	0.3022	1.00
2	0.1626	1.86
4	0.1019	2.97
6	0.0801	3.77
8	0.0731	4.13

Assemble Tangent Matrix

Yingrui (Ray) Chang
California Institute of Technology

function definition:

```
SparseMatrix assembleTangentMatrix(const Displacements&) const;
```

difficulties when working with sparse matrices:

- only stores nonzero elements;
- inserting new elements would involve linear time copying;
- usually constructing sparse matrix from *triplets* (linear time w.r.t the size of the triplets).

single thread matrix assembler (pseudo code):

```
//estimate #. nonzero entries (e.g. sparse level 1%)
initialize a global triplets
part 1:
for each element:
  get element displacements and node Id
  compute element tangent matrix
  figure out indices in the sparse matrix
  push indices and nonzero value into global triplets
part 2:
construct sparse matrix from the global triplets

return sparse matrix
```

sample run time in different parts:

part 1: 3.00s
part 2: 2.283s

parallelizing both parts is necessary.

Part2: Build Sparse Matrix from Triplets

Yingrui (Ray) Chang
California Institute of Technology

overall speedup factors when comparing to MATLAB:

Data Set Hardware	MATLAB Time	Serial Time	Serial Speedup	Parallel Time	Parallel Speedup
1 on C1	3.52	1.51	2.33×	0.65	5.39×
2 on C1	3.74	1.87	2.00×	0.83	4.42×
3 on C1	3.49	1.67	2.09×	0.76	4.55×
1 on C2	3.49	1.61	2.17×	0.33	10.2×
2 on C2	4.39	2.95	1.49×	0.46	9.71×
3 on C2	3.46	1.78	1.96×	0.43	9.01×

C1: 6 cores; C2: 16 cores.

references:

Fast MATLAB compatible sparse assembly on multicore computers, *Stefan Engblom and Dimtar Lukarski*

run time of parallel building sparse matrix from triplets:

#. of threads	run time (s)	speedup
1	2.282	1.00
2	1.148	1.99
3	0.799	2.86
4	0.627	3.64
5	0.506	4.50
6	0.438	5.21
7	0.389	5.86
8	0.353	6.46

Part1: Parallel Building of Triplets (Solution #1)

Yingrui (Ray) Chang
California Institute of Technology

similar idea as assemble energy:

- each thread writes to its local triplets list;
- once thread finishes its work, lock global triplets, pushes local triplet list into global triplets.

multithread Matrix Assembler (pseudo code):

```
initialize globalTriplets
#pragma omp parallel {
  initialize localTriplets
#pragma omp for
  for each element:
    get element displacements and node Id
    compute element tangent matrix
    figure out indices in the sparse matrix
    push indices and nonzero value into localTriplets
#pragma omp critial
  push localTriplets into global triplets
}
```

serialized copying worsens scaling

run time of parallel building triplets:

#. of threads	run time (s)	speedup
1	3.862	1.00
2	2.650	1.46
3	2.542	1.52
4	1.969	1.96
5	2.130	1.81
6	2.175	1.78
7	1.942	1.99
8	1.711	2.26

Part1: Parallel Building of Triplets (Solution #2)

Yingrui (Ray) Chang
California Institute of Technology

idea:

- parallelize the gathering of copies of triplets from each thread into a vector of triplets;
- get the total number of entries by summing the size of the triplets from the gathered vector;
- initialize global triplets with right number of entries
- parallel copy each of the local triplets into the right position

multithread Matrix Assembler (pseudo code):

```
initialize threadTripletsVector(numberOfThreads);
#pragma omp parallel {
  initialize localTriplets
#pragma omp for
  for each element:
    get element displacements and node Id
    compute element tangent matrix
    figure out indices in the sparse matrix
    push indices and nonzero value into localTriplets
#pragma omp critial
  threadTripletsVector.push_back(std::move(localTriplets))
}
compute total number of entries and initialize globalTriplets
#pragma omp parallel{
   ith thread copy the ith threadTriplets into globalTriplets
}
```

run time of parallel building of triplets:

#. of threads	run time (s)	speedup
1	3.329	1.00
2	1.754	1.89
3	1.293	2.57
4	1.072	3.11
5	0.902	3.69
6	0.801	4.15
7	0.730	4.56
8	0.666	5.00

Overall Performance of Assemble Tangent Matrix

Yingrui (Ray) Chang
California Institute of Technology

#. of threads	run time(s)	speedup
1	5.598	1.00
2	2.890	1.94
3	2.087	2.68
4	1.697	3.30
5	1.390	4.02
6	1.255	4.46
7	1.127	4.97
8	1.012	5.53

Summary & Conclusion

Yingrui (Ray) Chang
California Institute of Technology

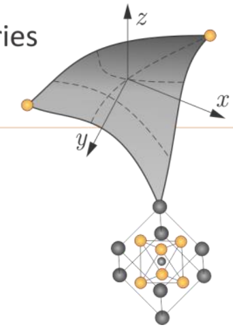
Template & Inheritance

- Designing finite element library by **templates** and **static programming techniques** could lead better performing, maintainable and safer code.
- Consider using template over inheritance (virtual function) approach unless constraints on recompilation time and size of executables are crucial.

Performance

- Threading both **solver** and **assembler** are necessary to utilize **multi core machine** to build scalable program.
- **Critical sessions** should be used **with caution** in order to reach better scalability. General guide line would be only allow **constant time operations** in critical sessions.

Graduate Aerospace Laboratories
Kochmann Research Group



Thank you for your interest!

Questions & Comments

Yingrui (Ray) Chang
yingryic@gmail.com
www.its.caltech.edu/~ycchang

Acknowledgements:

Dennis Kochmann, Jeff Amelang, Ishan Tembhekar, Alex Zelhofer

Support from the **Army Research Labs** through the **MEDE Center** is gratefully acknowledged.