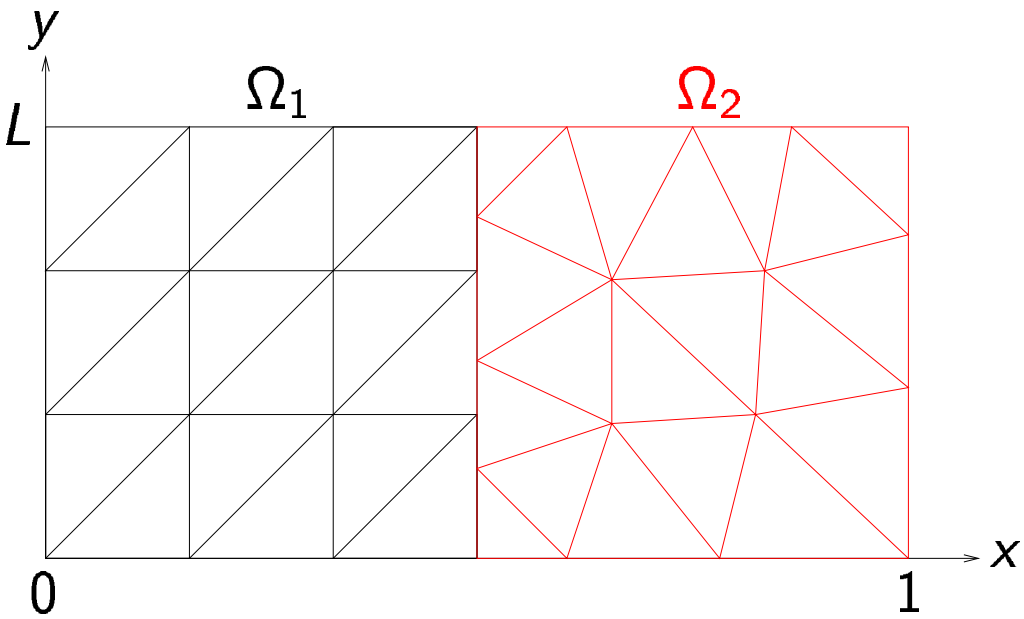
An Algorithm with Optimal Complexity for Non-Matching Grid Projections

Martin J. Gander
martin.gander@math.unige.ch

University of Geneva

February, 2009

Mortar Methods: Coupling Non-Matching Grids



Bernardi, Maday and Patera: Domain decomposition by the mortar element method (1993)

Transmission conditions for the model problem $(\eta - \Delta)u = f$:

$$\partial_x u_1 + p u_1 = \partial_x u_2 + p u_2, \quad \partial_x u_2 - p u_2 = \partial_x u_1 - p u_1$$

Discretized variational formulation contains integrals of the form

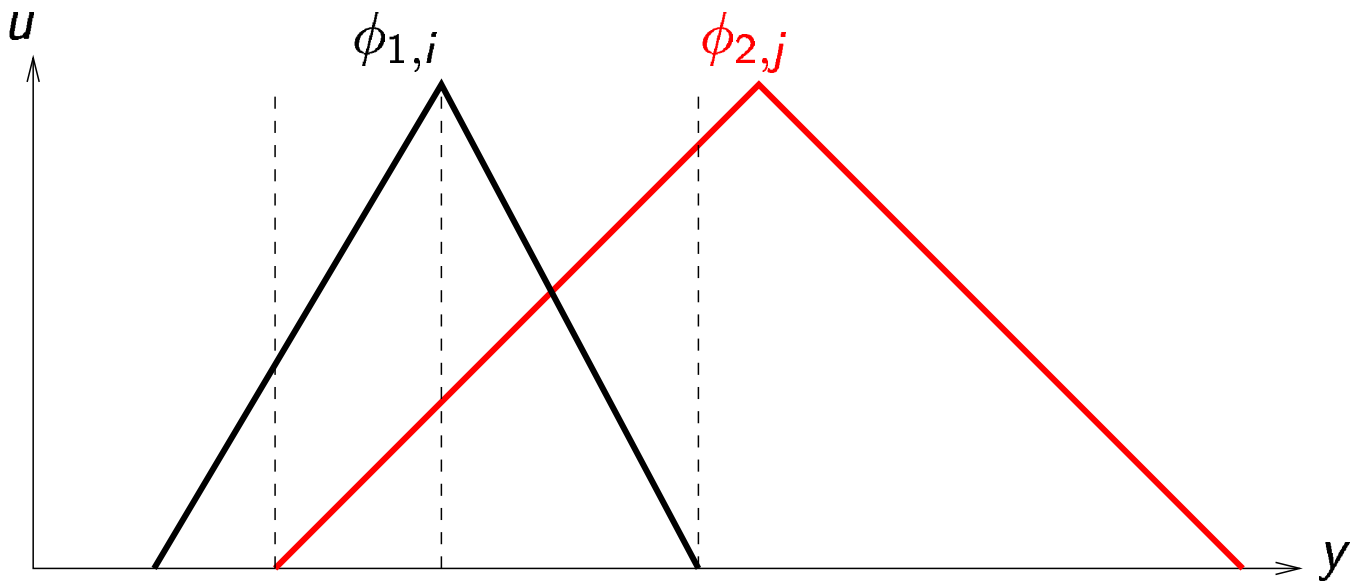
$$\int_0^L \phi_{1,i} \phi_{2,j} dy, \qquad \phi_{1,i},\ \phi_{2,j} \text{ hat functions on } \Omega_1 \text{ and } \Omega_2.$$

Coupling in the Mortar Method

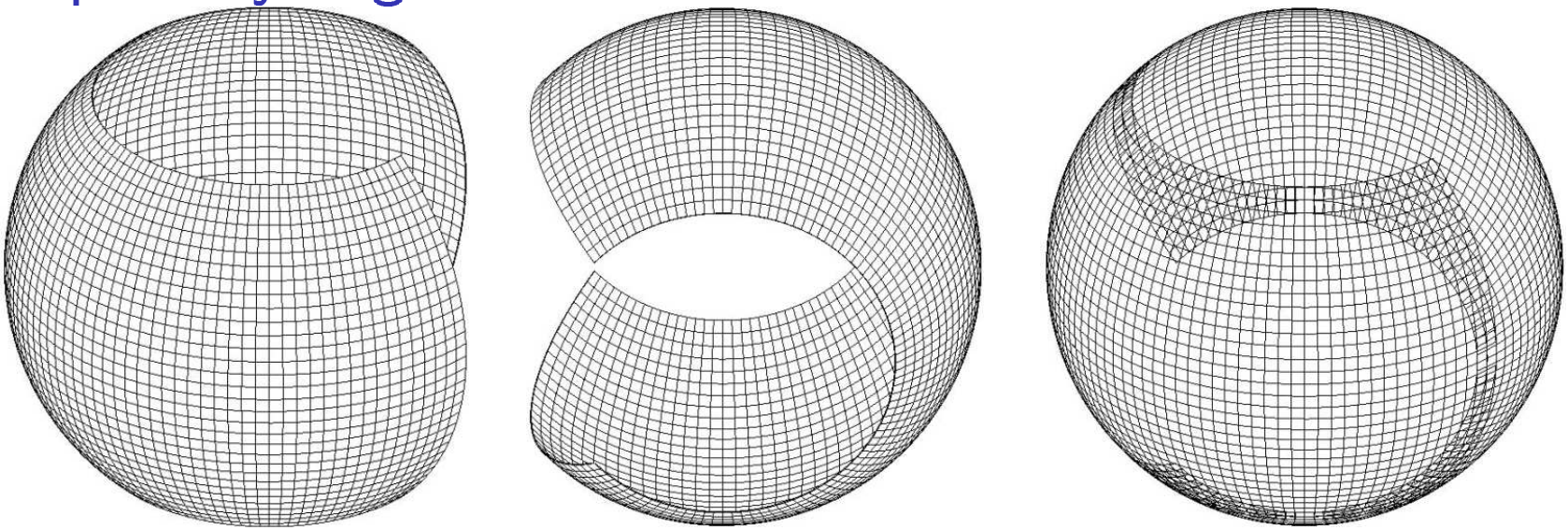
One needs to compute the coupling matrix

$$M_{ij} := \int_0^L \phi_{1,i}\phi_{2,j}dy,$$

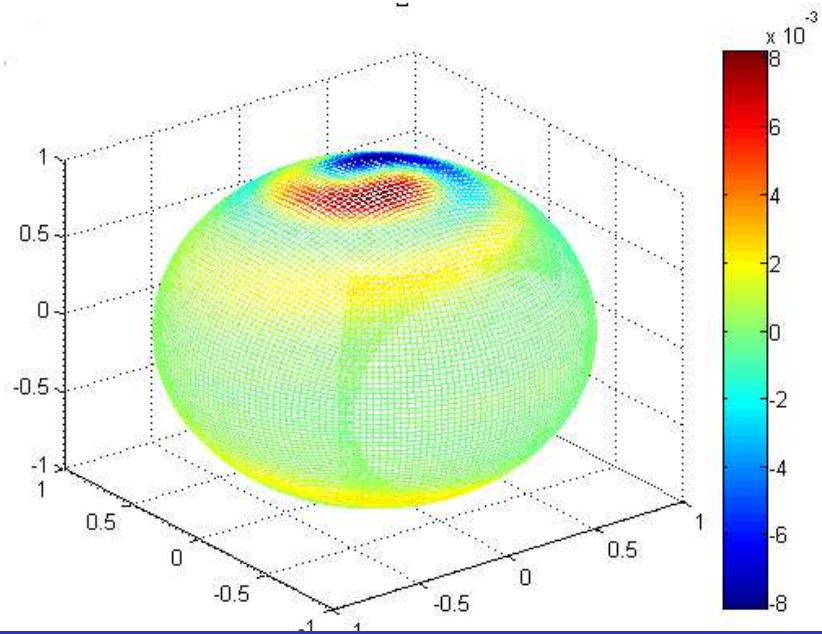
where $\phi_{1,i}$ and $\phi_{2,j}$ are hat functions on the interface associated with the non-matching grids of Ω_1 and Ω_2.



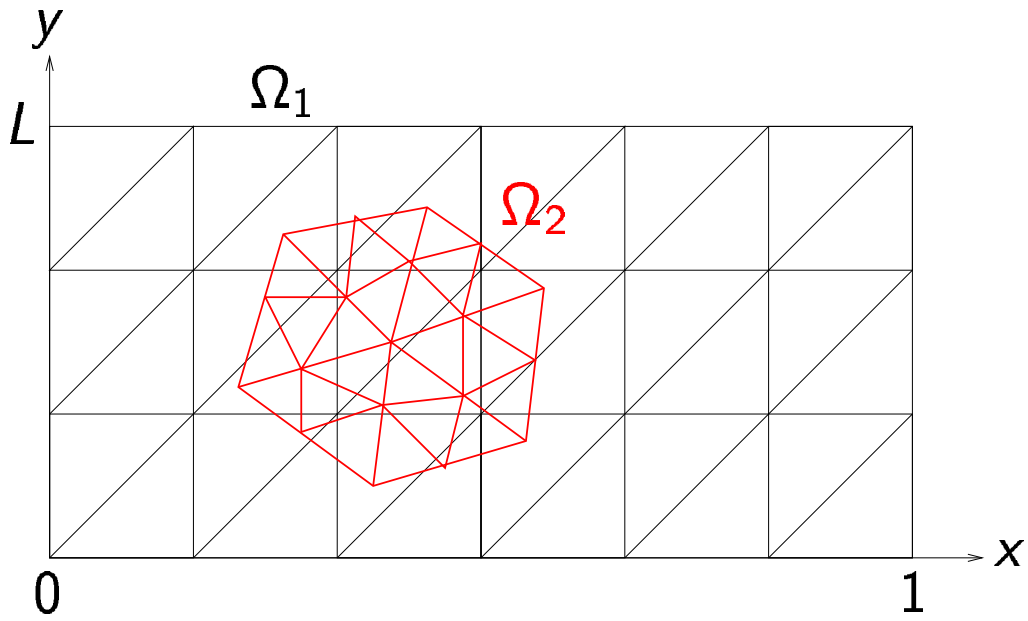
Example: Cyclogenesis Test



In global weather simulations, the flow computations are performed on the entire globe. In order to avoid pole problems in the mesh, a Yin-Yang grid can be used (with Côté and Qaddouri 2006)



Patch Methods: Numerical Zoom



Glowinski, He, Lozinski, Rappaz and Wagner: finite element approximation of multi-scale elliptic problems using patches of elements (2005)

For $(\eta - \Delta)u = f$ and two finite element spaces V_1 and V_2, the 'patch algorithm' computes for $u^0 \in V_1$ and $\omega \in (0,2)$

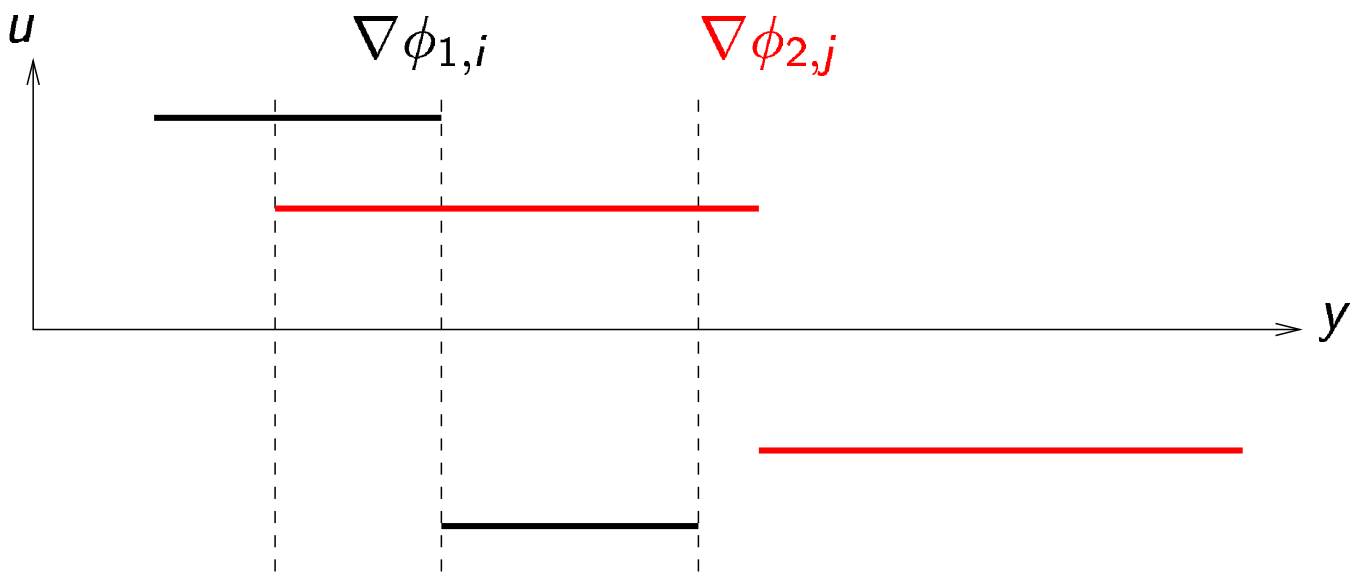
$$w_2 \in V_2 \text{ s.t. } a(w_2, v) = (f, v) - a(u^{n-1}, v), \ \forall v \in V_2$$
$$u^{n-\frac{1}{2}} = u^{n-1} + \omega w_2$$
$$w_1 \in V_1 \text{ s.t. } a(w_1, v) = (f, v) - a(u^{n-\frac{1}{2}}, v), \ \forall v \in V_1$$
$$u^{n} = u^{n-\frac{1}{2}} + \omega w_1$$

Coupling in the Patch Method

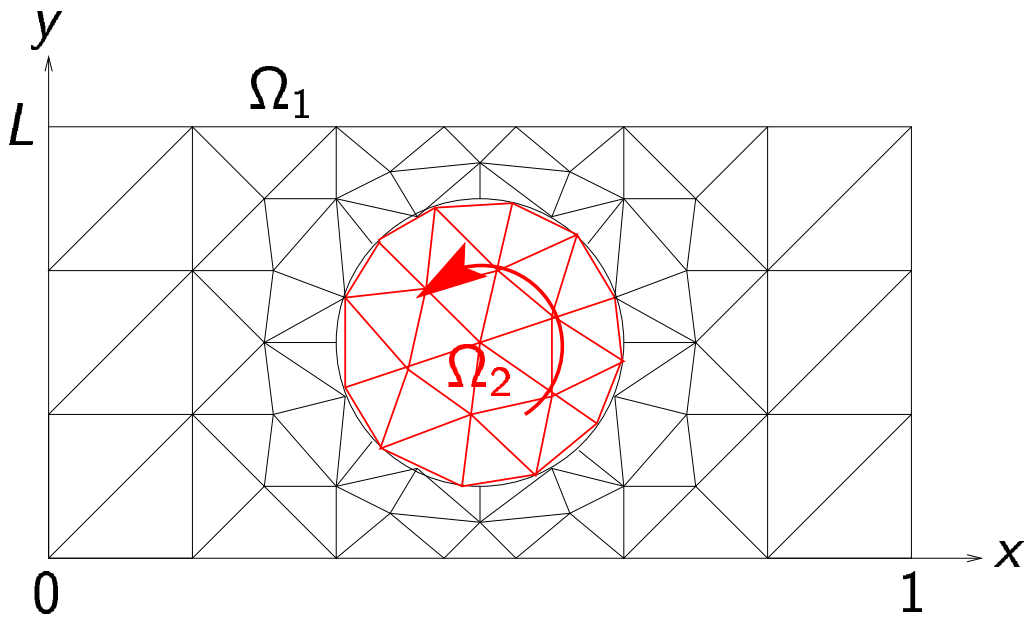
In patch methods, we need to compute the coupling matrix

$$M_{ij} := \int_P \nabla \phi_{1,i} \cdot \nabla \phi_{2,j} ds$$

where $\phi_{1,i}$ are hat functions associated with the global grid of domain Ω_1, and $\phi_{2,j}$ are hat functions of the local patch grid on Ω_2.



Chimera Methods



Domain decomposition method for problems with moving parts:

- Steger, Dougherty and Benek: A Chimera Grid Scheme, Advances in Grid Generation (1983),
- Brezzi, Lions and Pironneau: Analysis of a Chimera method (2001)

Difficulty 1: Algorithmic Complexity

Flemisch, Kaltenbacher and Wohlmuth: Elasto-Acoustic and Acoustic-Acoustic Coupling on Nonmatching Grids (2002).

> *One possible realization is given in Algorithm 1; we remark that the naive implementation of this algorithm for 2d-problems is of order $O(n^2)$.*

```
for i=1:I                                  % Algorithm 1
  for j=1:J
    P=intersect(i,j);
    if ~isempty(P)
      M(i,j)=integrate(i,j,P);
    end
  end;
end;
```

Every element could intersect with every other element !

Difficulty 2: Numerical Computation of Intersections

Flemisch, Kaltenbacher and Wohlmuth: Elasto-Acoustic and Acoustic-Acoustic Coupling on Nonmatching Grids (2002).

> *The intersection of two arbitrary triangles is quite more complex than the situation in 1d.*

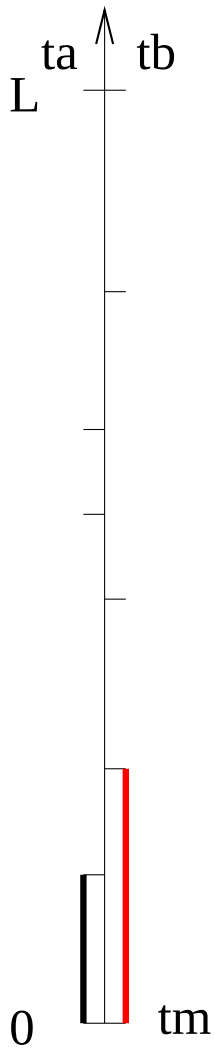
Picasso, Rappaz, Rezzonico: Multiscale Algorithm with Patches of Finite Elements (2007)

> *Some difficulties remain though since we must compute integrals involving shape functions that are defined on non-compatible meshes.*

Maday, Rapetti, Wohlmuth: The Influence of Quadrature Formulas in 2d and 3d Mortar Element Methods (2007)

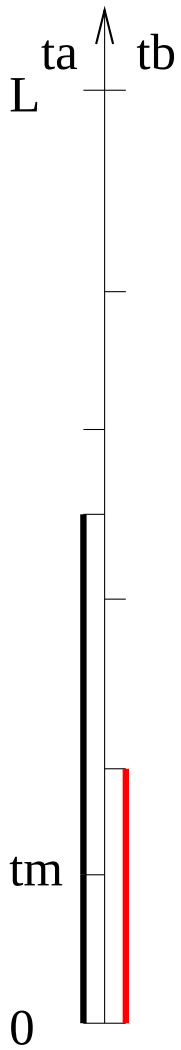
An Optimal Algorithm in 1d

```
ta and tb given partition vectors;

j=1;
for i=1:length(tb)-1
  tm=tb(i);
  while ta(j+1)<tb(i+1)
    M(i,j)=integrate(i,j,tm,ta(j+1));
    j=j+1;
    tm=ta(j);
  end;
  M(i,j)=integrate(i,j,tm,tb(i+1));
end;
```



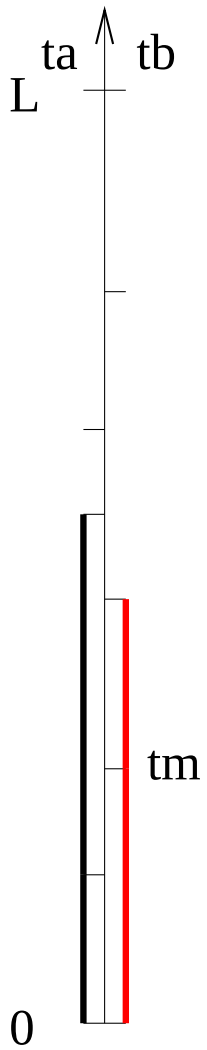
An Optimal Algorithm: step 2

```
ta and tb given partitions;

j=1;
for i=1:length(tb)-1
  tm=tb(i);
  while ta(j+1)<tb(i+1)
    M(i,j)=integrate(i,j,tm,ta(j+1));
    j=j+1;
    tm=ta(j);
  end;
  M(i,j)=integrate(i,j,tm,tb(i+1));
end;
```



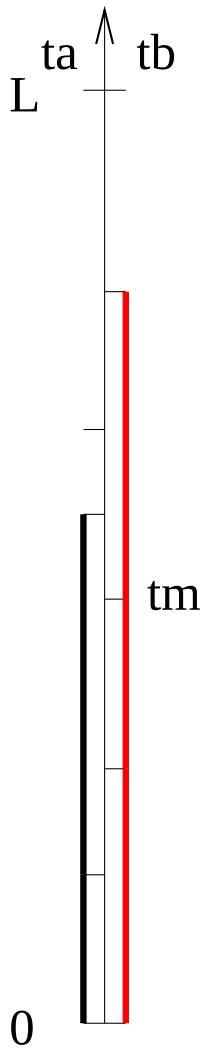
An Optimal Algorithm: step 3

```
ta and tb given partitions;

j=1;
for i=1:length(tb)-1
  tm=tb(i);
  while ta(j+1)<tb(i+1)
    M(i,j)=integrate(i,j,tm,ta(j+1));
    j=j+1;
    tm=ta(j);
  end;
  M(i,j)=integrate(i,j,tm,tb(i+1));
end;
```



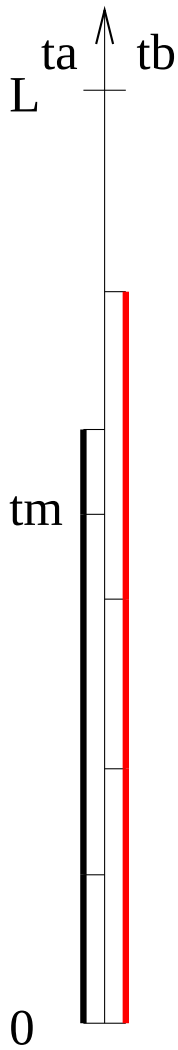
An Optimal Algorithm: step 4

```
ta and tb given partitions;

j=1;
for i=1:length(tb)-1
  tm=tb(i);
  while ta(j+1)<tb(i+1)
    M(i,j)=integrate(i,j,tm,ta(j+1));
    j=j+1;
    tm=ta(j);
  end;
  M(i,j)=integrate(i,j,tm,tb(i+1));
end;
```



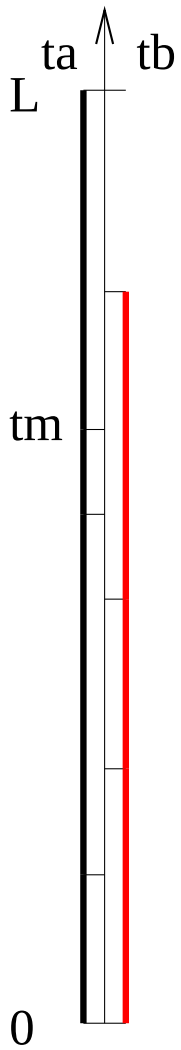
An Optimal Algorithm: step 5

```
ta and tb given partitions;

j=1;
for i=1:length(tb)-1
  tm=tb(i);
  while ta(j+1)<tb(i+1)
    M(i,j)=integrate(i,j,tm,ta(j+1));
    j=j+1;
    tm=ta(j);
  end;
  M(i,j)=integrate(i,j,tm,tb(i+1));
end;
```



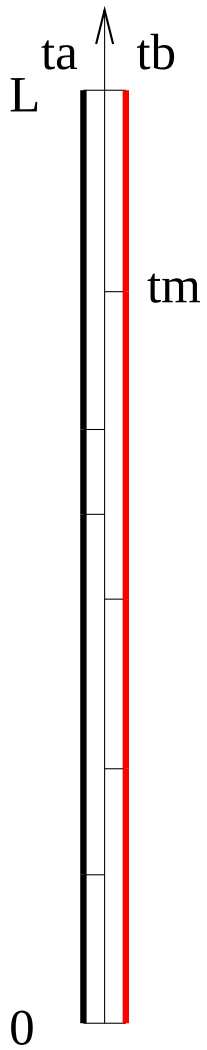
An Optimal Algorithm: step 6

```
ta and tb given partitions;

j=1;
for i=1:length(tb)-1
  tm=tb(i);
  while ta(j+1)<tb(i+1)
    M(i,j)=integrate(i,j,tm,ta(j+1));
    j=j+1;
    tm=ta(j);
  end;
  M(i,j)=integrate(i,j,tm,tb(i+1));
end;
```



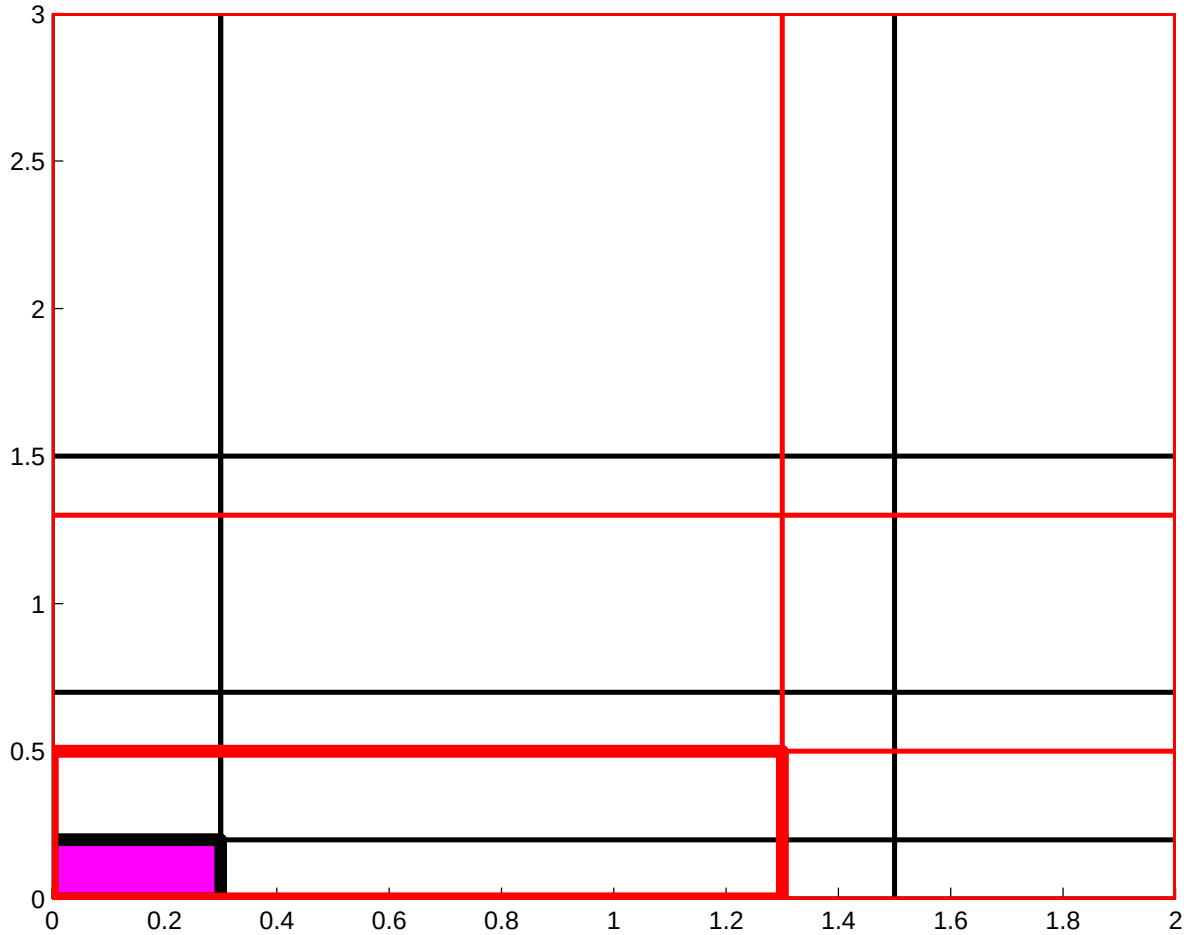
An Optimal Algorithm: last step

```
ta and tb given partitions;

j=1;
for i=1:length(tb)-1
  tm=tb(i);
  while ta(j+1)<tb(i+1)
    M(i,j)=integrate(i,j,tm,ta(j+1));
    j=j+1;
    tm=ta(j);
  end;
  M(i,j)=integrate(i,j,tm,tb(i+1));
end;
```

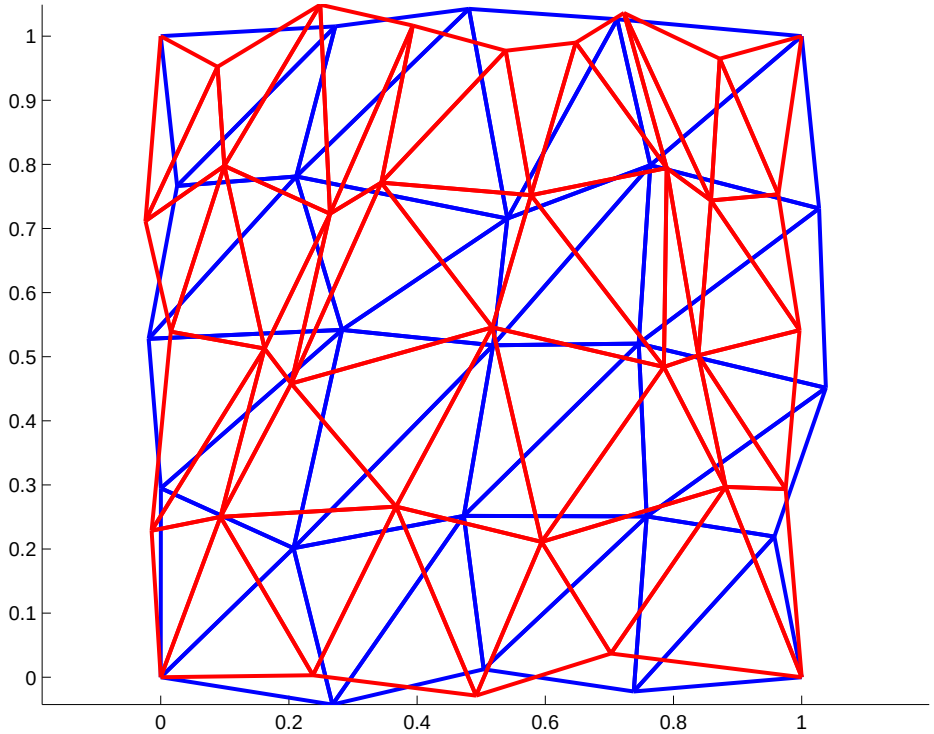


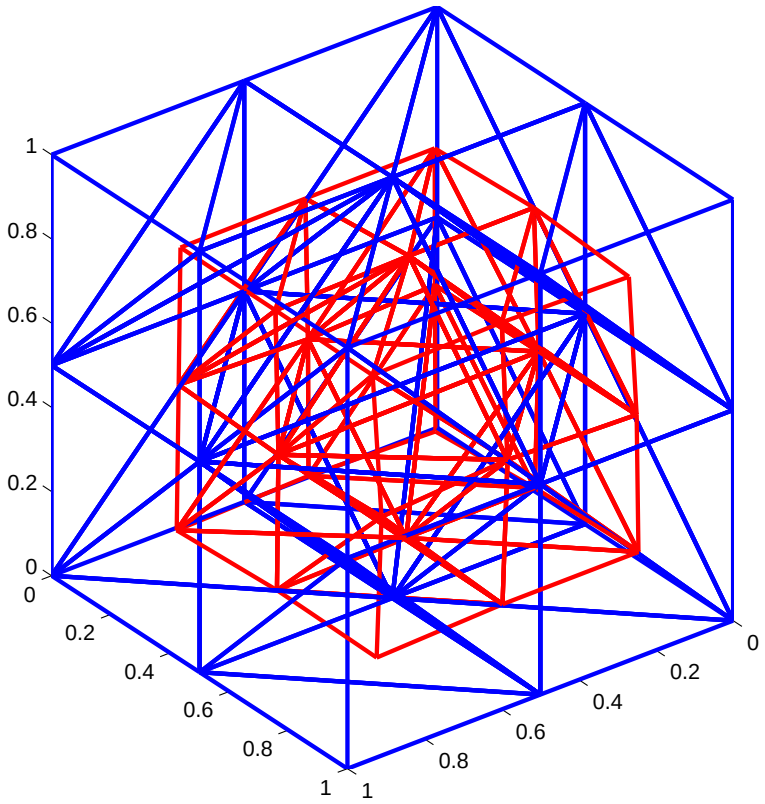
Tensor Product Mesh in 2d

Use the optimal 1d algorithm in both directions simultaneously.



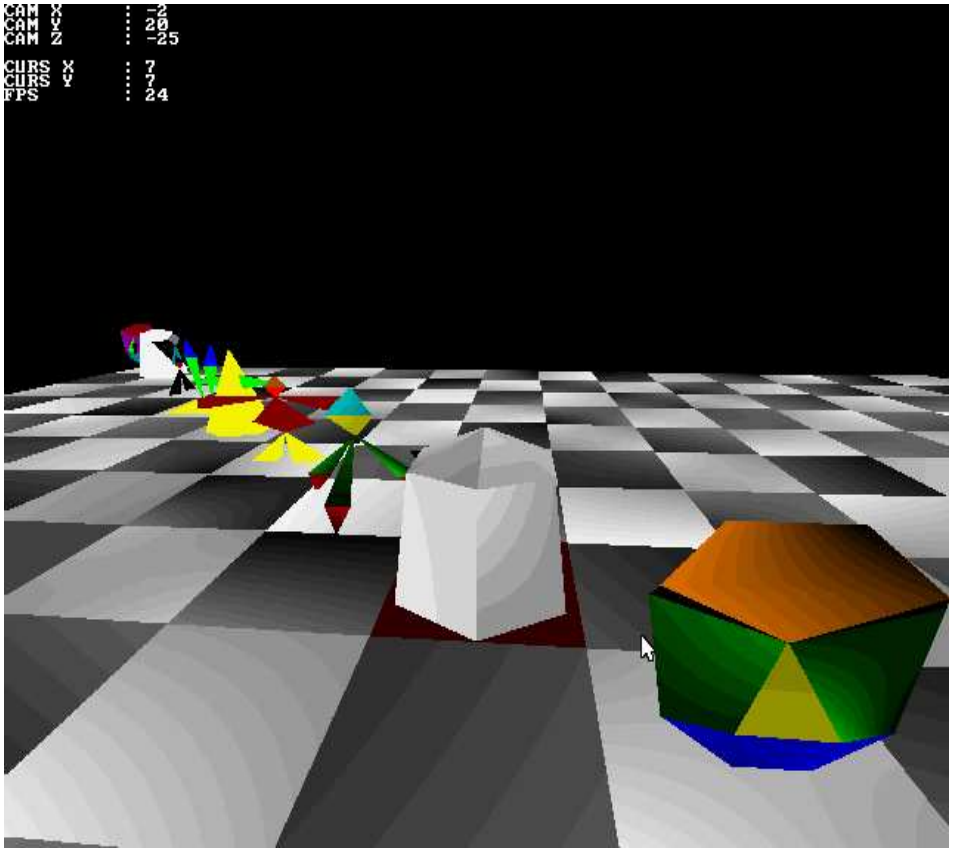
General Case and Higher Dimensions





1. How to compute intersections ?
2. How to get optimal complexity ?

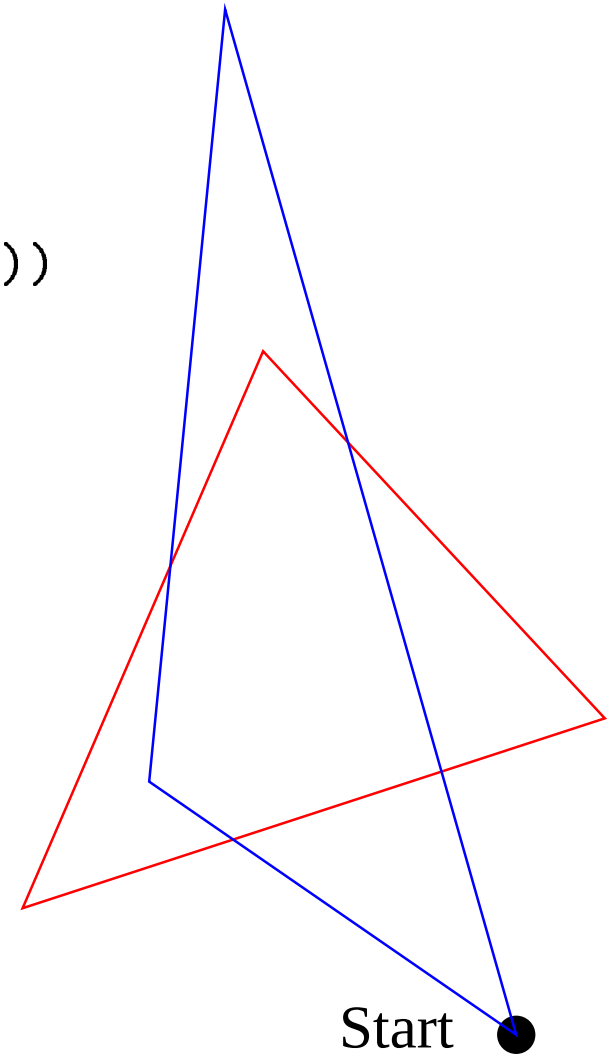
Intersection Algorithms from Computer Graphics



- Weiler and Atherton: Hidden surface removal using polygon area sorting (1977)
- Greiner and Hormann: Efficient clipping of arbitrary polygons (1998)

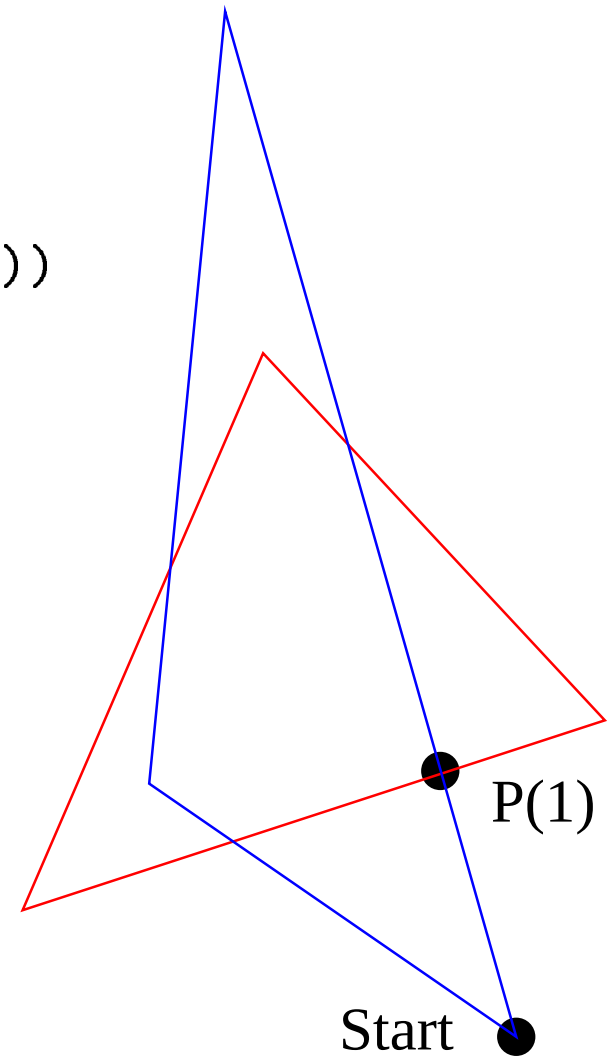
Computing the intersection: polygon clipping

```
> [Start,T1,T2]=OutsideCorner(T1,T2);
  P(1)=GetNextPoint(Start,T1,T2);
  i=1;
  while i<2 | (P(i)<>Start & P(i)<>P(1))
    if i>1 & isIntersection(P(i))
      exchange T1 and T2;
    end
    P(i+1)=GetNextPoint(P(i),T1,T2);
    i=i+1;
  end
  if P(i)==Start & IsIn(T2,T1)
    P=T2;
  else
    P=[];
  end;
```



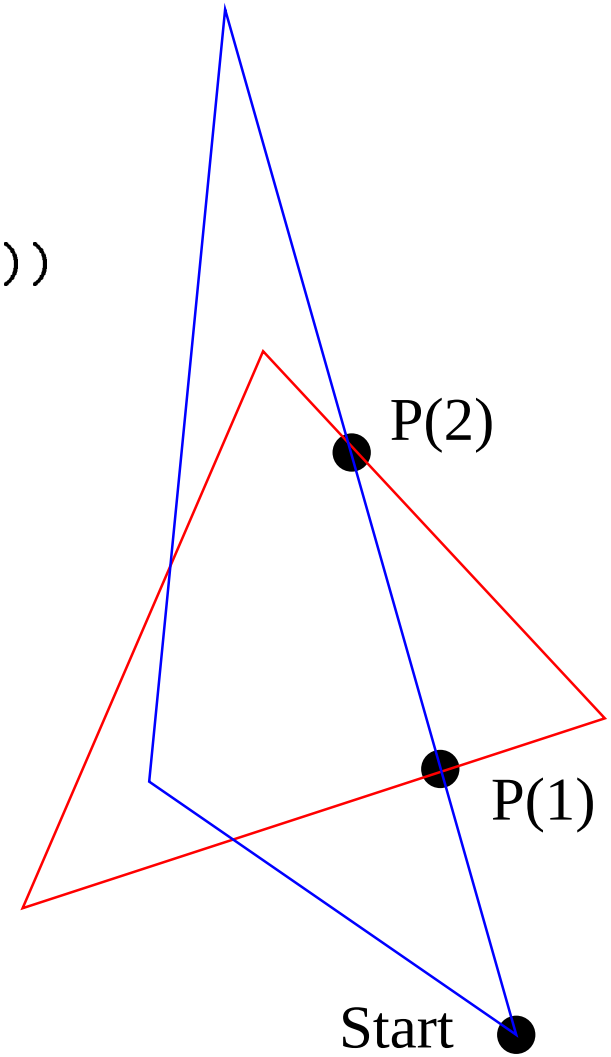
Computing the intersection: step 1

```
  [Start,T1,T2]=OutsideCorner(T1,T2);
> P(1)=GetNextPoint(Start,T1,T2);
  i=1;
  while i<2 | (P(i)<>Start & P(i)<>P(1))
    if i>1 & isIntersection(P(i))
      exchange T1 and T2;
    end
    P(i+1)=GetNextPoint(P(i),T1,T2);
    i=i+1;
  end
  if P(i)==Start & IsIn(T2,T1)
    P=T2;
  else
    P=[];
  end;
```



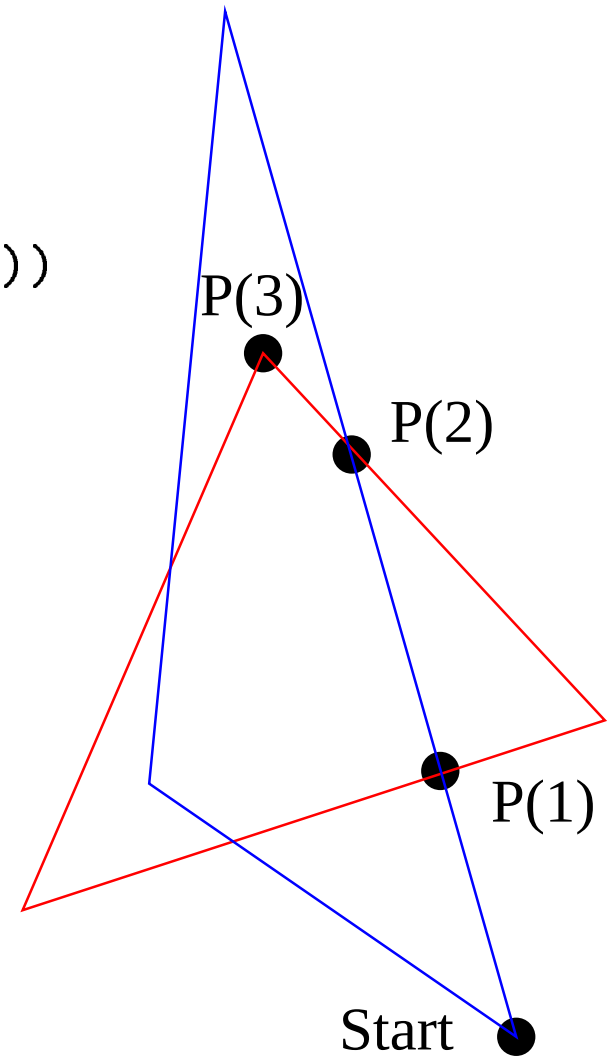
Computing the intersection: step 2

```
  [Start,T1,T2]=OutsideCorner(T1,T2);
  P(1)=GetNextPoint(Start,T1,T2);
  i=1;
> while i<2 | (P(i)<>Start & P(i)<>P(1))
    if i>1 & isIntersection(P(i))
      exchange T1 and T2;
    end
    P(i+1)=GetNextPoint(P(i),T1,T2);
    i=i+1;
  end
  if P(i)==Start & IsIn(T2,T1)
    P=T2;
  else
    P=[];
  end;
```



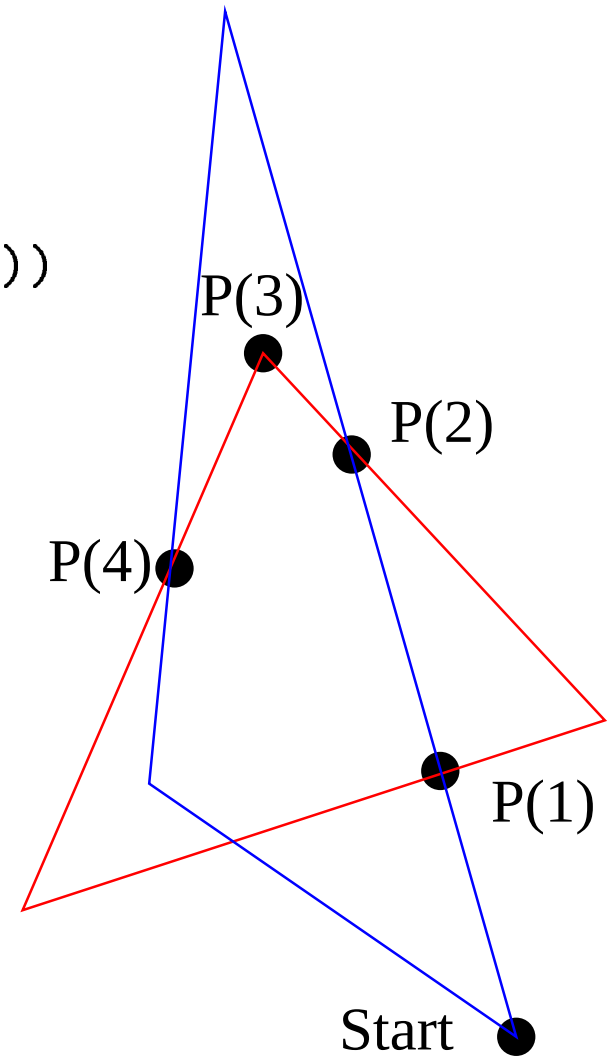
Computing the intersection: step 3

```
  [Start,T1,T2]=OutsideCorner(T1,T2);
  P(1)=GetNextPoint(Start,T1,T2);
  i=1;
> while i<2 | (P(i)<>Start & P(i)<>P(1))
    if i>1 & isIntersection(P(i))
      exchange T1 and T2;
    end
    P(i+1)=GetNextPoint(P(i),T1,T2);
    i=i+1;
  end
  if P(i)==Start & IsIn(T2,T1)
    P=T2;
  else
    P=[];
  end;
```



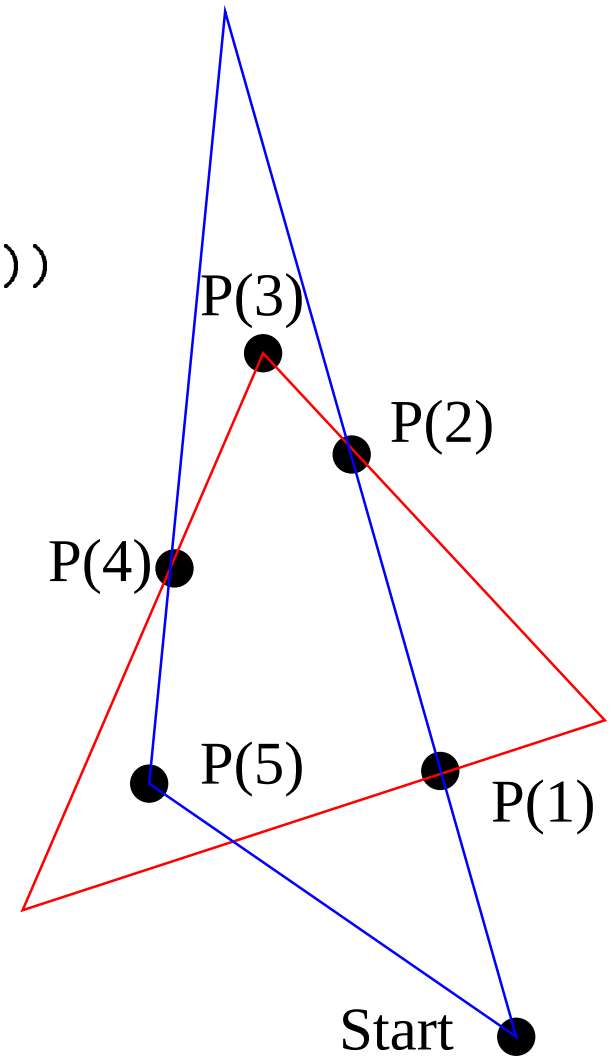
Computing the intersection: step 4

```
  [Start,T1,T2]=OutsideCorner(T1,T2);
  P(1)=GetNextPoint(Start,T1,T2);
  i=1;
> while i<2 | (P(i)<>Start & P(i)<>P(1))
    if i>1 & isIntersection(P(i))
      exchange T1 and T2;
    end
    P(i+1)=GetNextPoint(P(i),T1,T2);
    i=i+1;
  end
  if P(i)==Start & IsIn(T2,T1)
    P=T2;
  else
    P=[];
  end;
```



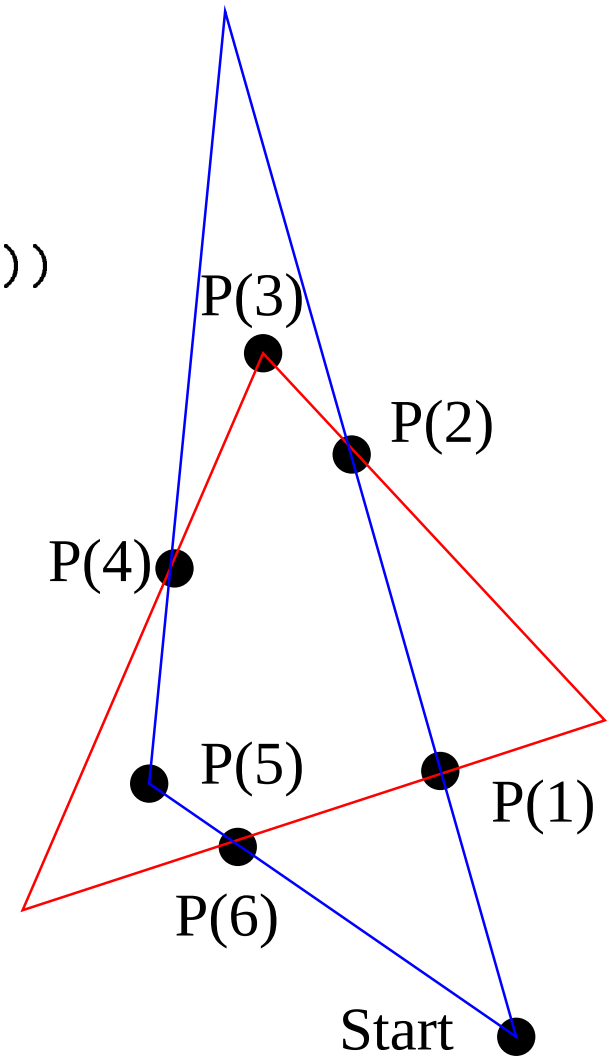
Computing the intersection: step 5

```
  [Start,T1,T2]=OutsideCorner(T1,T2);
  P(1)=GetNextPoint(Start,T1,T2);
  i=1;
> while i<2 | (P(i)<>Start & P(i)<>P(1))
    if i>1 & isIntersection(P(i))
      exchange T1 and T2;
    end
    P(i+1)=GetNextPoint(P(i),T1,T2);
    i=i+1;
  end
  if P(i)==Start & IsIn(T2,T1)
    P=T2;
  else
    P=[];
  end;
```



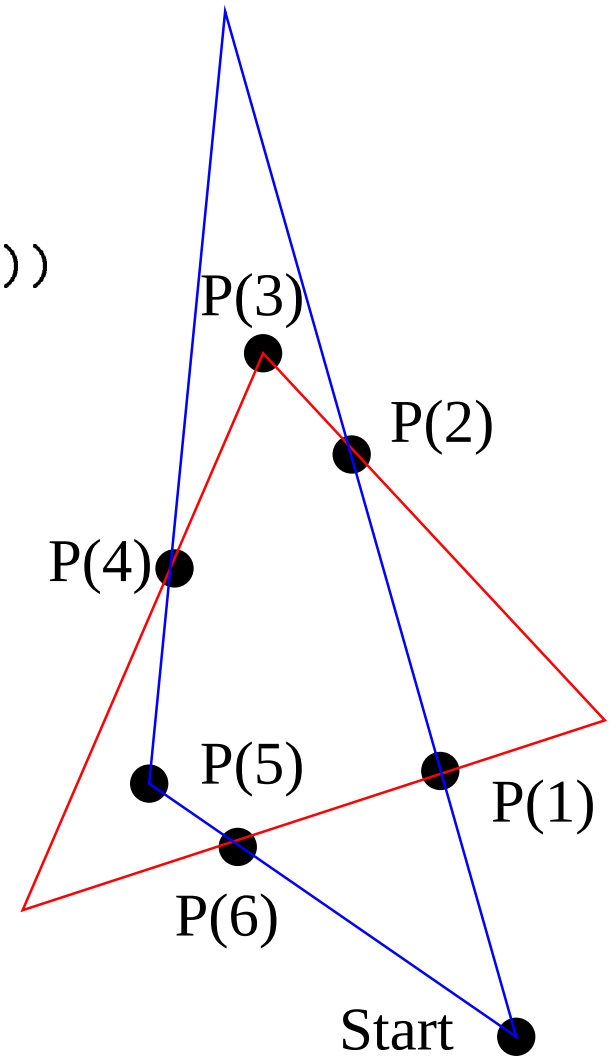
Computing the intersection: step 6

```
  [Start,T1,T2]=OutsideCorner(T1,T2);
  P(1)=GetNextPoint(Start,T1,T2);
  i=1;
> while i<2 | (P(i)<>Start & P(i)<>P(1))
    if i>1 & isIntersection(P(i))
      exchange T1 and T2;
    end
    P(i+1)=GetNextPoint(P(i),T1,T2);
    i=i+1;
  end
  if P(i)==Start & IsIn(T2,T1)
    P=T2;
  else
    P=[];
  end;
```

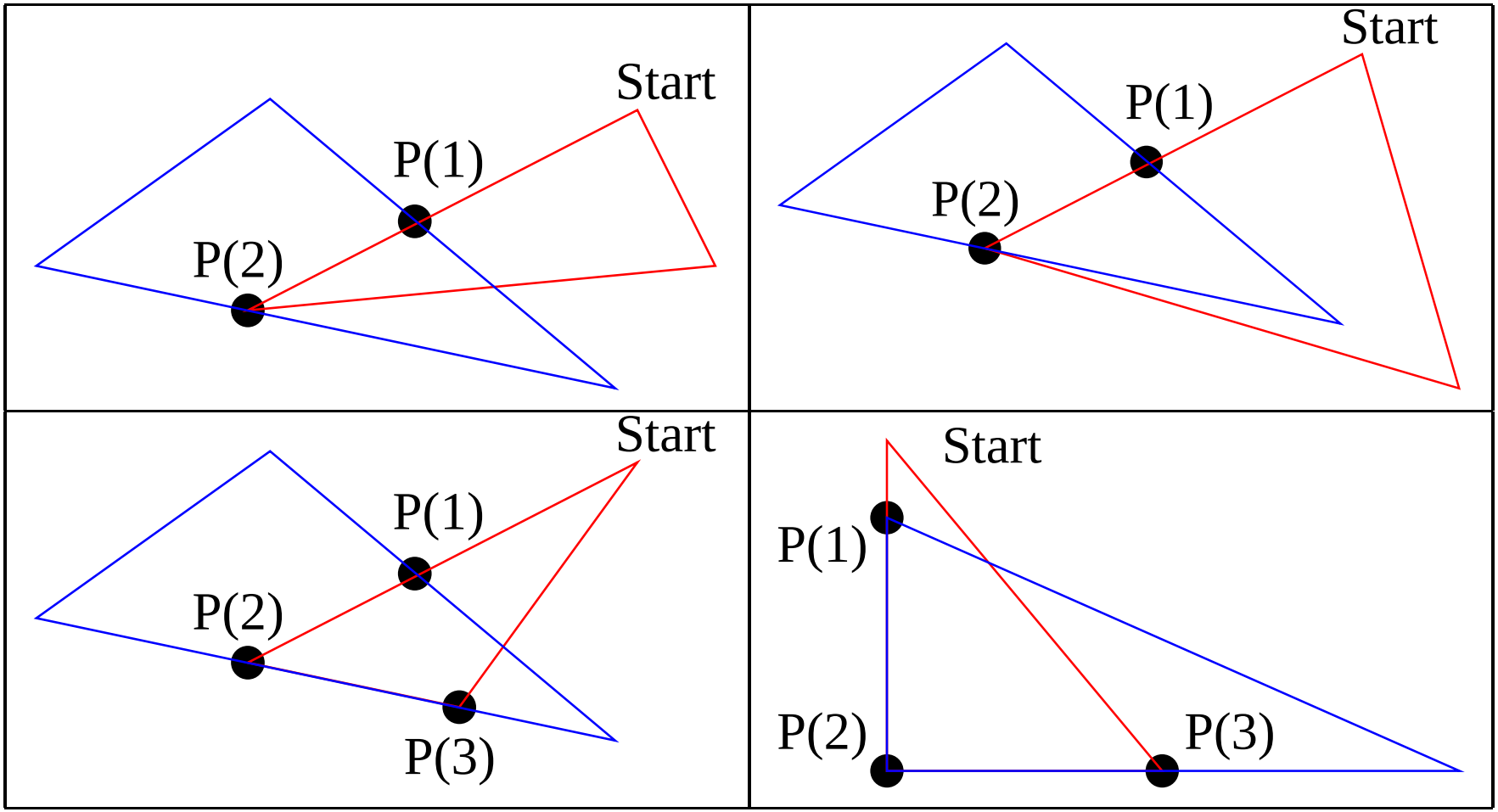


Computing the intersection: step 7

```
  [Start,T1,T2]=OutsideCorner(T1,T2);
  P(1)=GetNextPoint(Start,T1,T2);
  i=1;
  while i<2 | (P(i)<>Start & P(i)<>P(1))
    if i>1 & isIntersection(P(i))
      exchange T1 and T2;
    end
    P(i+1)=GetNextPoint(P(i),T1,T2);
    i=i+1;
  end
> if P(i)==Start & IsIn(T2,T1)
    P=T2;
  else
    P=[];
  end;
```



Problems in Polygon Clipping



Solutions in the Literature

Greiner and Hormann, Efficient Clipping of Arbitrary Polygons (1998).

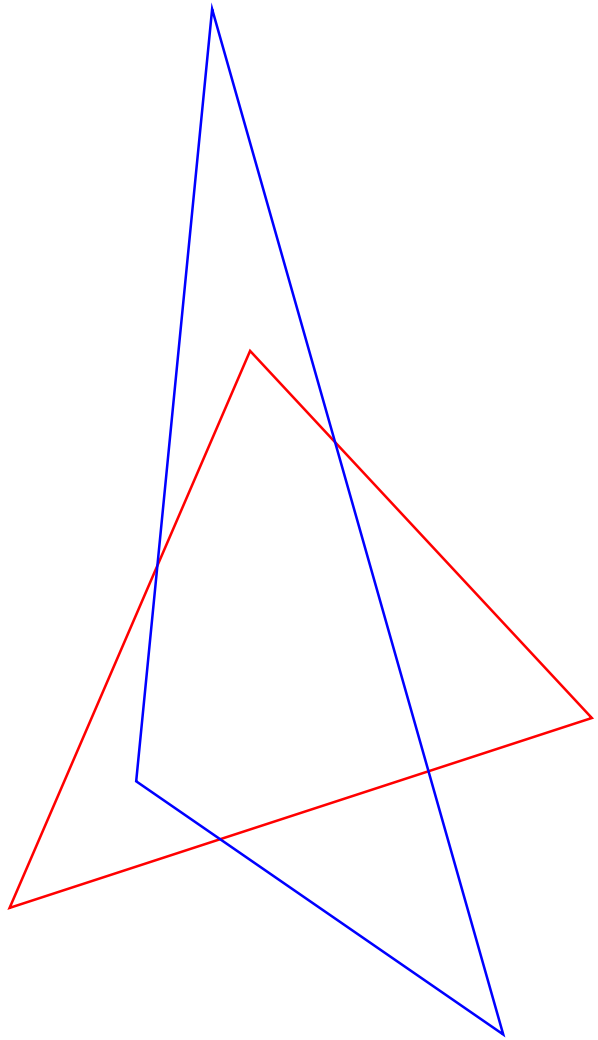
> *So far we have tacitly assumed that there are no* degeneracies, *i.e., each vertex of one polygon does not lie on an edge of the other polygon. Degeneracies can be detected in the intersect procedure. . . . In this case, we perturb P slightly . . . If we take care that the perturbation is less than a pixel width, the output on the screen will be correct.*

Shewchuk, Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates, 1996.

> *Use higher precision arithmetic to make numerical decisions in difficult cases.*

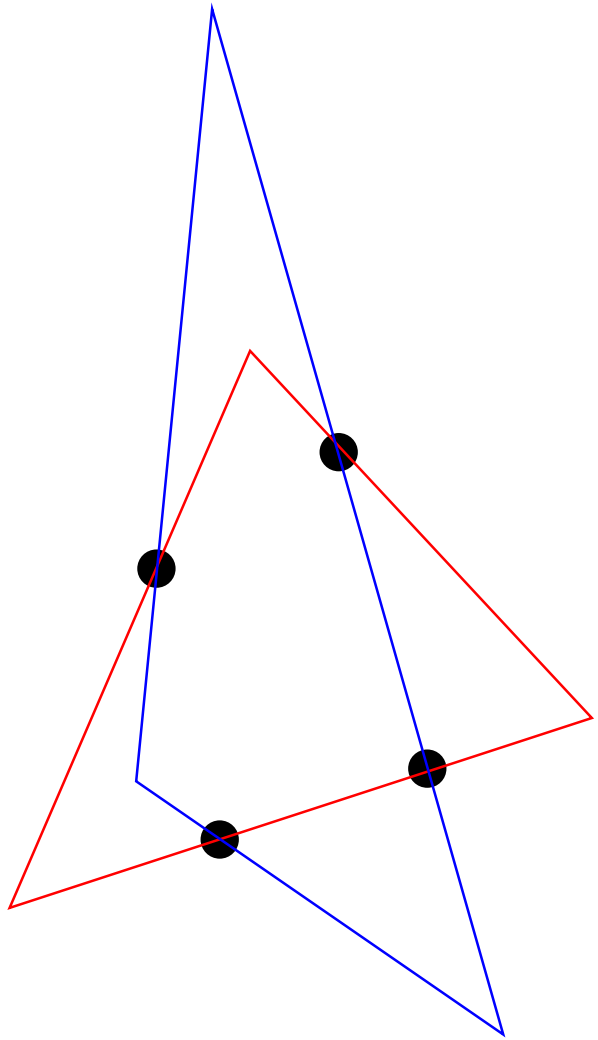
Computing the intersection: a new approach

```
P=EdgeIntersections(T1,T2);
P=[P PointsIn(T1,T2)];
P=[P PointsIn(T2,T1)];
P=SortCounterclockwise(P);
P=RemoveDuplicates(P,eps);
```



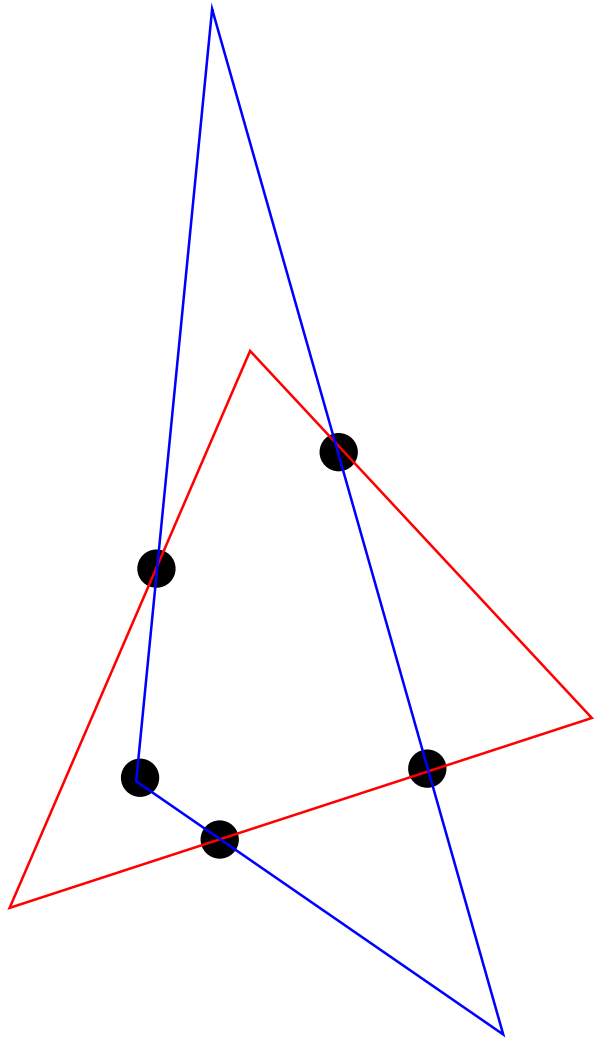
Computing the intersection: step 1

```
> P=EdgeIntersections(T1,T2);
  P=[P PointsIn(T1,T2)];
  P=[P PointsIn(T2,T1)];
  P=SortCounterclockwise(P);
  P=RemoveDuplicates(P,eps);
```



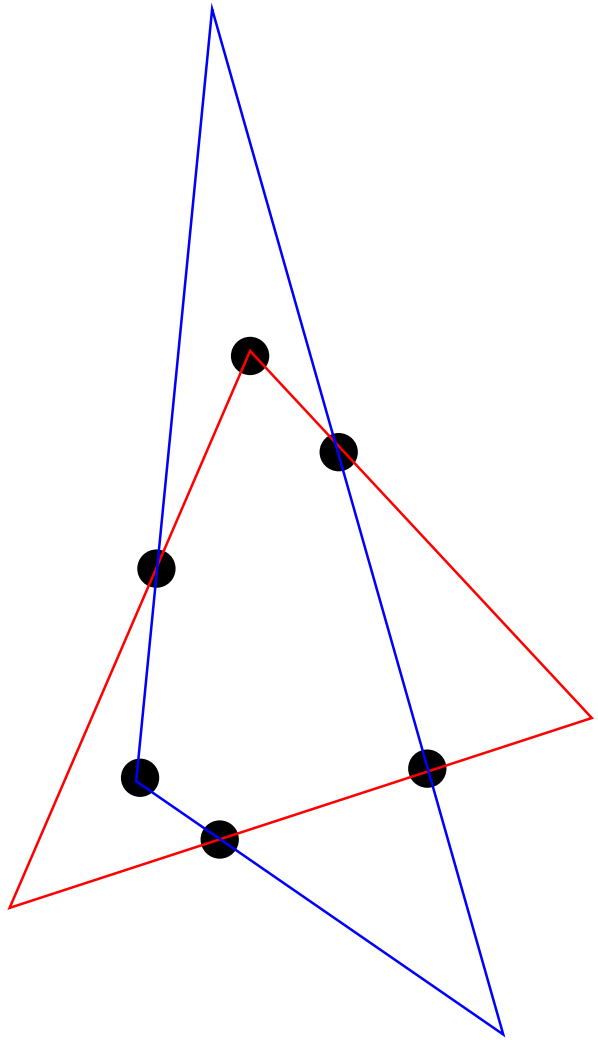
Computing the intersection: step 2

```
  P=EdgeIntersections(T1,T2);
> P=[P PointsIn(T1,T2)];
  P=[P PointsIn(T2,T1)];
  P=SortCounterclockwise(P);
  P=RemoveDuplicates(P,eps);
```



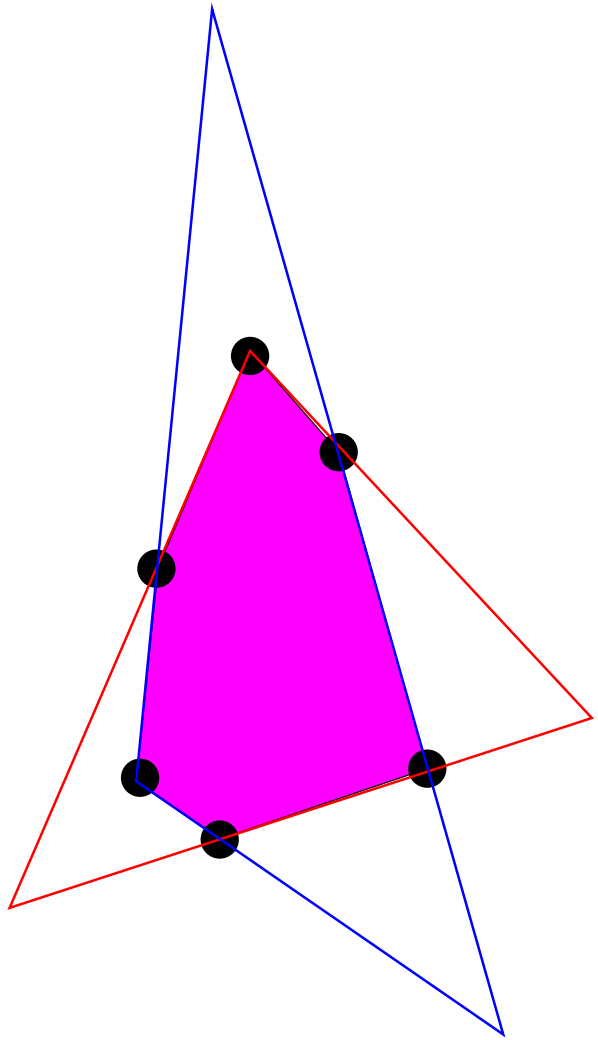
Computing the intersection: step 3

```
  P=EdgeIntersections(T1,T2);
  P=[P PointsIn(T1,T2)];
> P=[P PointsIn(T2,T1)];
  P=SortCounterclockwise(P);
  P=RemoveDuplicates(P,eps);
```



Computing the intersection: step 3

```
  P=EdgeIntersections(T1,T2);
  P=[P PointsIn(T1,T2)];
  P=[P PointsIn(T2,T1)];
> P=SortCounterclockwise(P);
> P=RemoveDuplicates(P,eps);
```

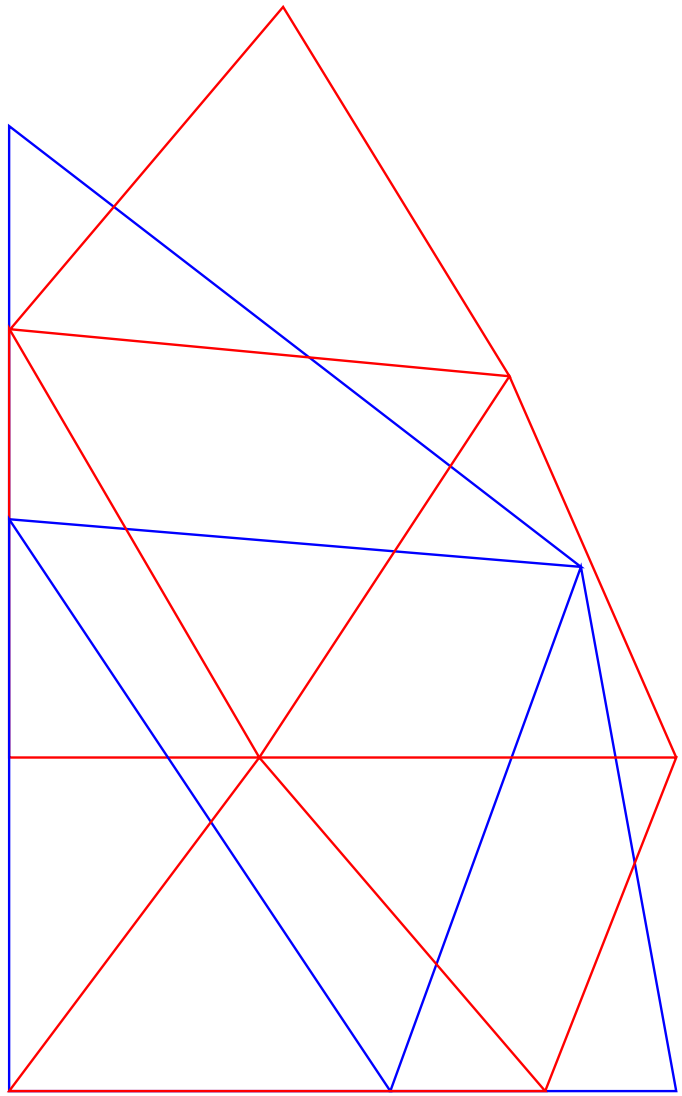


Reducing the Complexity

- Knuth (1973): for interpolation, using a refined background quadtree mesh, $O(n \log n)$.
- Löhner and Morgan (1987): binning (or bucket) technique.
- Löhner (2001): for interpolation, using neighboring element information, an advancing front technique for the points where the interpolation has to be performed, and a vicinity search, almost linear complexity.
- Lee, C. Yang and J. Yang (2004): self-avoiding walk with vicinity search, approximately linear complexity
- Plimpton, Hendrickson and Stewart: A parallel rendezvous algorithm for interpolation between multiple grids (2004)

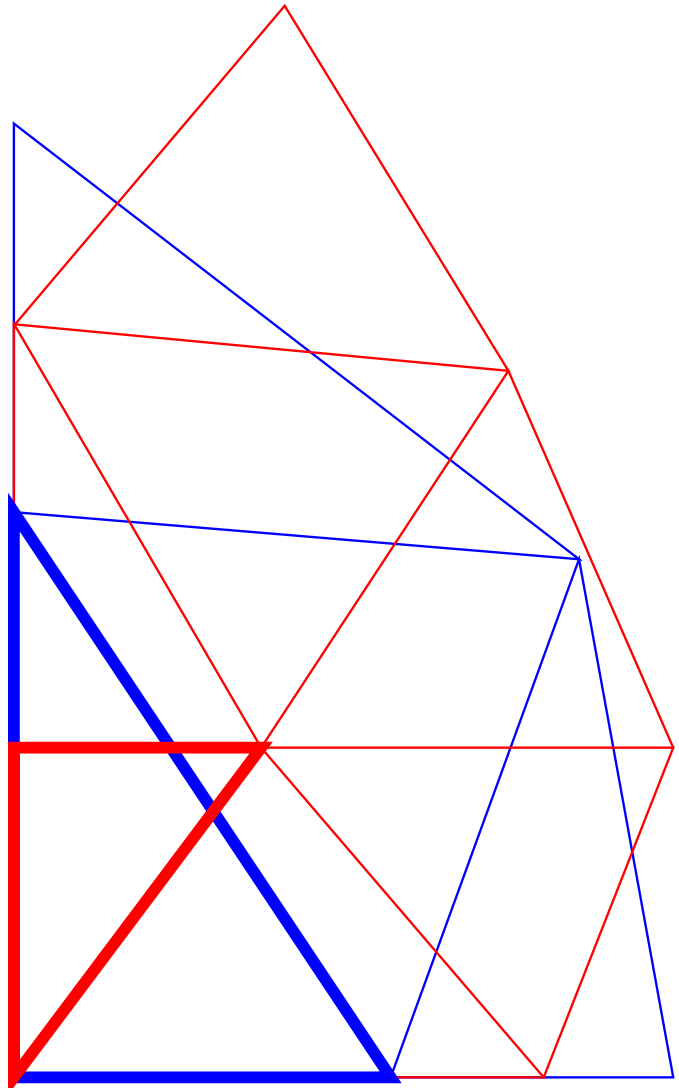
Our Algorithm with Optimal Complexity in n-dimensions

```
Ta and Tb two lists of simplices;
bl=simplex of Tb;
bil=simplex of Ta intersecting bl;
while length(bl)>0
  bc=bl(1); bl=bl(2:end);
  al=bil(1); bil=bil(2:end);
  while length(al)>0
    ac=al(1);  al=al(2:end);
    [P,nc,Mc]=integrate(ac,bc);
    if ~isempty(P)
      M(ac,bc)=M(ac,bc)+Mc;
      al=[al neighbors(ac)];
      n(nc)=ac;
    end;
  end;
  bl=[bl untreatedneighbors(bc)];
  bil=[bil n(untreatedneighbors(bc))];
end;
```



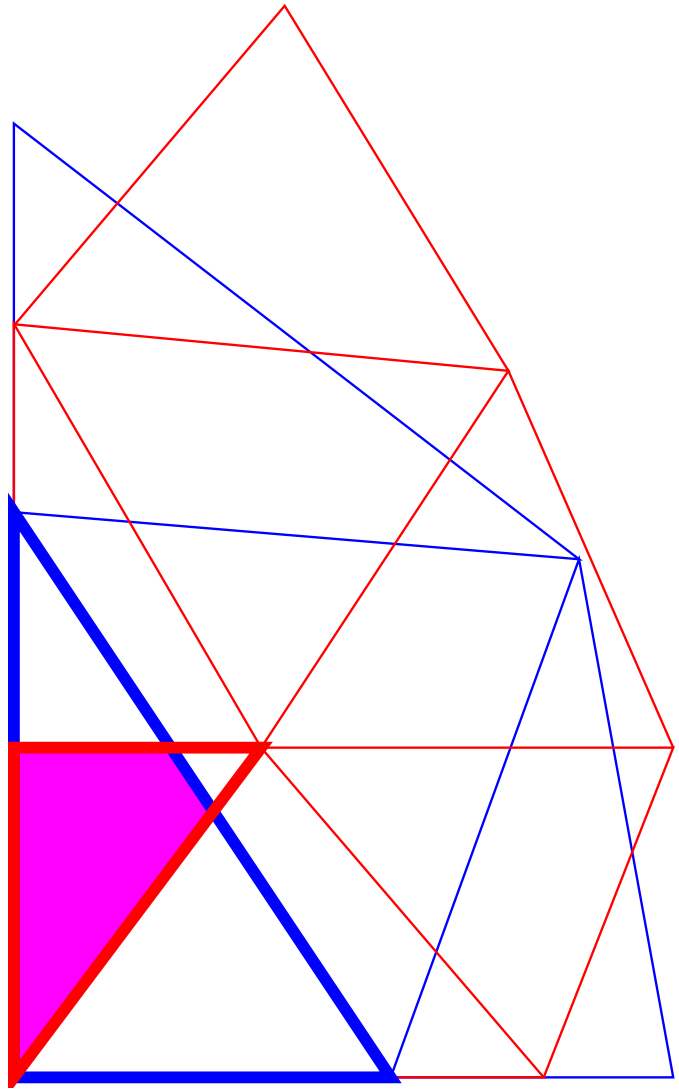
Start with two Intersecting Triangles

```
  Ta and Tb two lists of simplices;
  bl=simplex of Tb;
  bil=simplex of Ta intersecting bl;
  while length(bl)>0
>   bc=bl(1); bl=bl(2:end);
>   al=bil(1); bil=bil(2:end);
    while length(al)>0
      ac=al(1);  al=al(2:end);
      [P,nc,Mc]=integrate(ac,bc);
      if ~isempty(P)
        M(ac,bc)=M(ac,bc)+Mc;
        al=[al neighbors(ac)];
        n(nc)=ac;
      end;
    end;
    bl=[bl untreatedneighbors(bc)];
    bil=[bil n(untreatedneighbors(bc))];
  end;
```



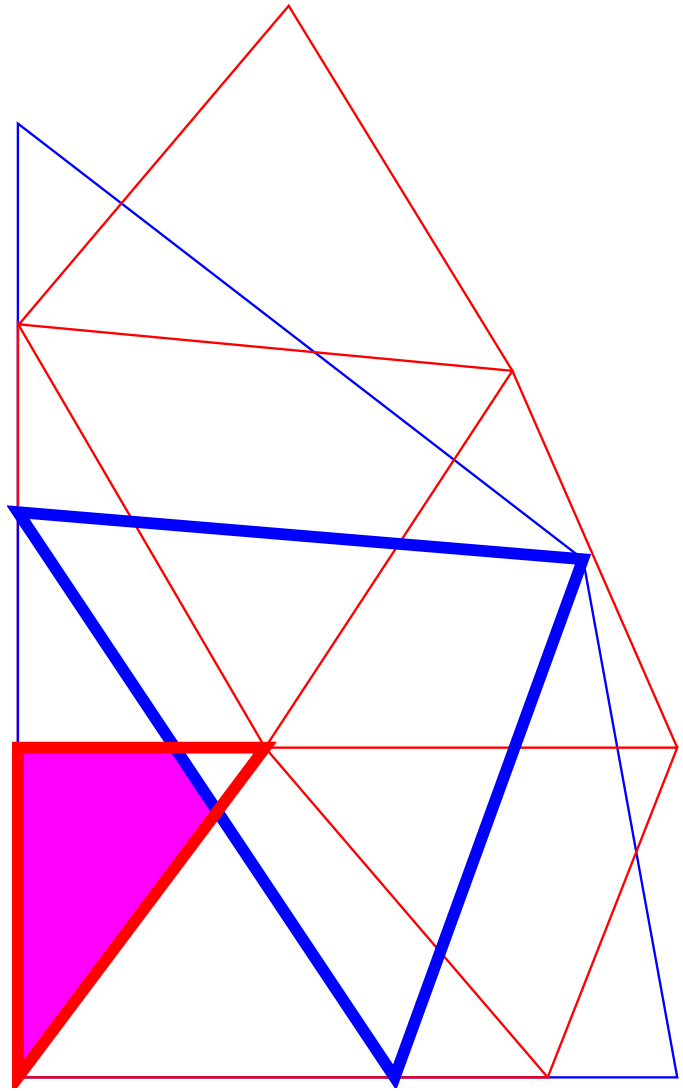
Compute Their Intersection

```
  Ta and Tb two lists of simplices;
  bl=simplex of Tb;
  bil=simplex of Ta intersecting bl;
  while length(bl)>0
    bc=bl(1); bl=bl(2:end);
    al=bil(1); bil=bil(2:end);
    while length(al)>0
      ac=al(1);  al=al(2:end);
>     [P,nc,Mc]=integrate(ac,bc);
      if ~isempty(P)
        M(ac,bc)=M(ac,bc)+Mc;
        al=[al neighbors(ac)];
        n(nc)=ac;
      end;
    end;
    bl=[bl untreatedneighbors(bc)];
    bil=[bil n(untreatedneighbors(bc))];
  end;
```



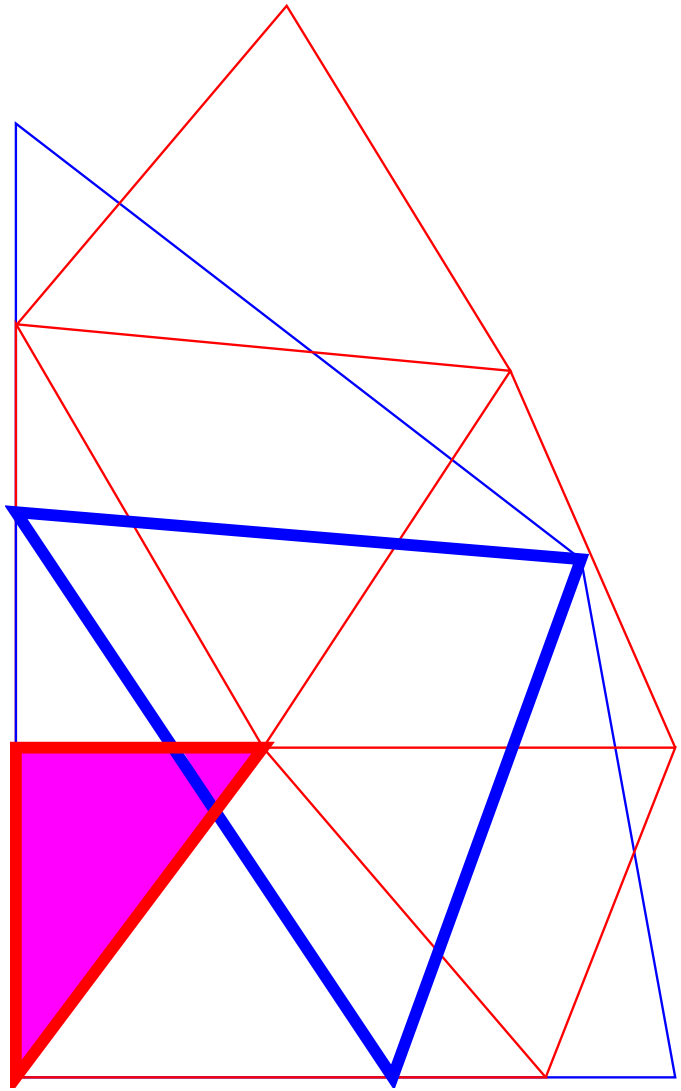
Add Untreated Neighbors, Remember Candidates

```
Ta and Tb two lists of simplices;
bl=simplex of Tb;
bil=simplex of Ta intersecting bl;
while length(bl)>0
  bc=bl(1); bl=bl(2:end);
  al=bil(1); bil=bil(2:end);
  while length(al)>0
    ac=al(1);  al=al(2:end);
    [P,nc,Mc]=integrate(ac,bc);
    if ~isempty(P)
      M(ac,bc)=M(ac,bc)+Mc;
>     al=[al neighbors(ac)];
>     n(nc)=ac;
    end;
  end;
  bl=[bl untreatedneighbors(bc)];
  bil=[bil n(untreatedneighbors(bc))];
end;
```



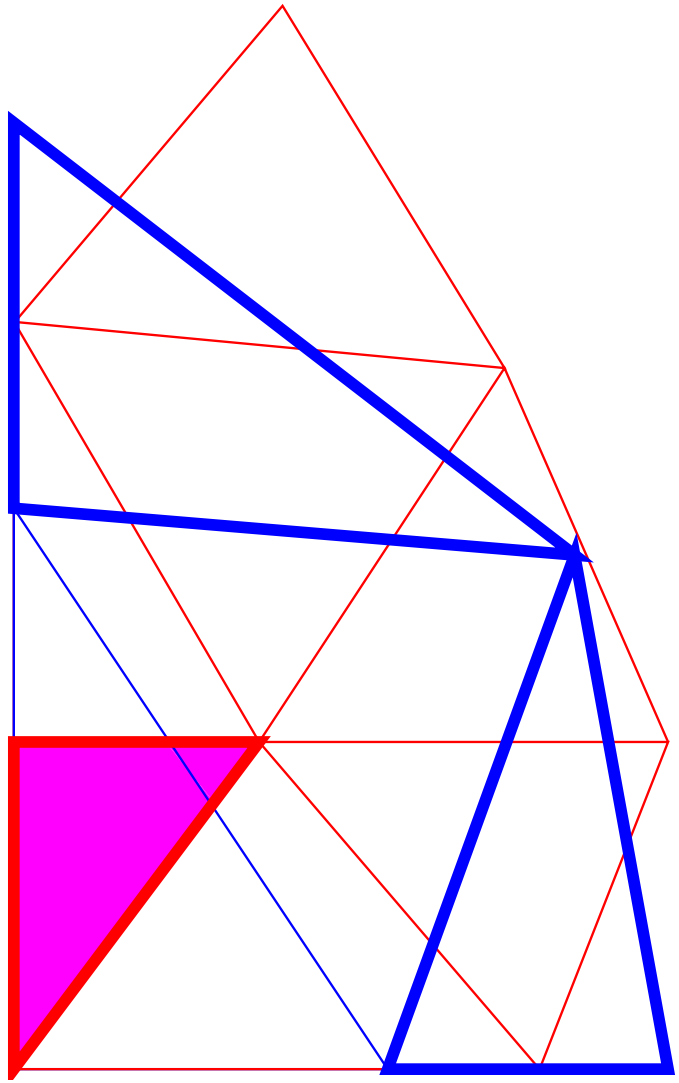
Compute Intersection

```
  Ta and Tb two lists of simplices;
  bl=simplex of Tb;
  bil=simplex of Ta intersecting bl;
  while length(bl)>0
    bc=bl(1); bl=bl(2:end);
    al=bil(1); bil=bil(2:end);
    while length(al)>0
      ac=al(1);  al=al(2:end);
>     [P,nc,Mc]=integrate(ac,bc);
      if ~isempty(P)
        M(ac,bc)=M(ac,bc)+Mc;
        al=[al neighbors(ac)];
        n(nc)=ac;
      end;
    end;
    bl=[bl untreatedneighbors(bc)];
    bil=[bil n(untreatedneighbors(bc))];
  end;
```



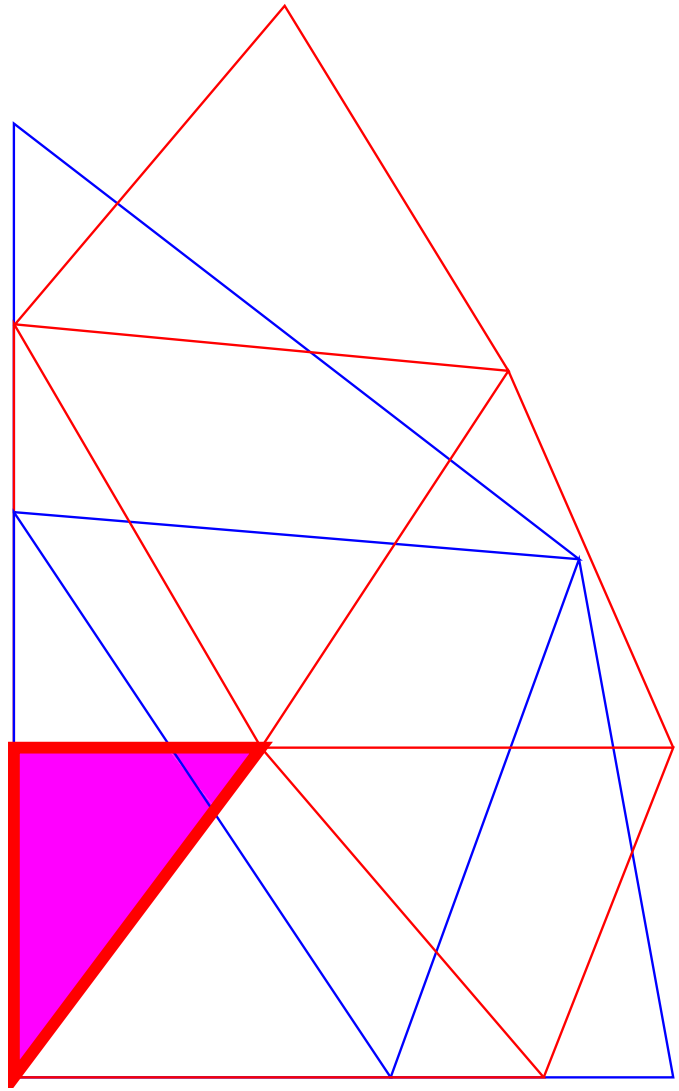
Add Untreated Neighbors

```
Ta and Tb two lists of simplices;
bl=simplex of Tb;
bil=simplex of Ta intersecting bl;
while length(bl)>0
  bc=bl(1); bl=bl(2:end);
  al=bil(1); bil=bil(2:end);
  while length(al)>0
    ac=al(1);  al=al(2:end);
    [P,nc,Mc]=integrate(ac,bc);
    if ~isempty(P)
      M(ac,bc)=M(ac,bc)+Mc;
>     al=[al neighbors(ac)];
      n(nc)=ac;
    end;
  end;
  bl=[bl untreatedneighbors(bc)];
  bil=[bil n(untreatedneighbors(bc))];
end;
```



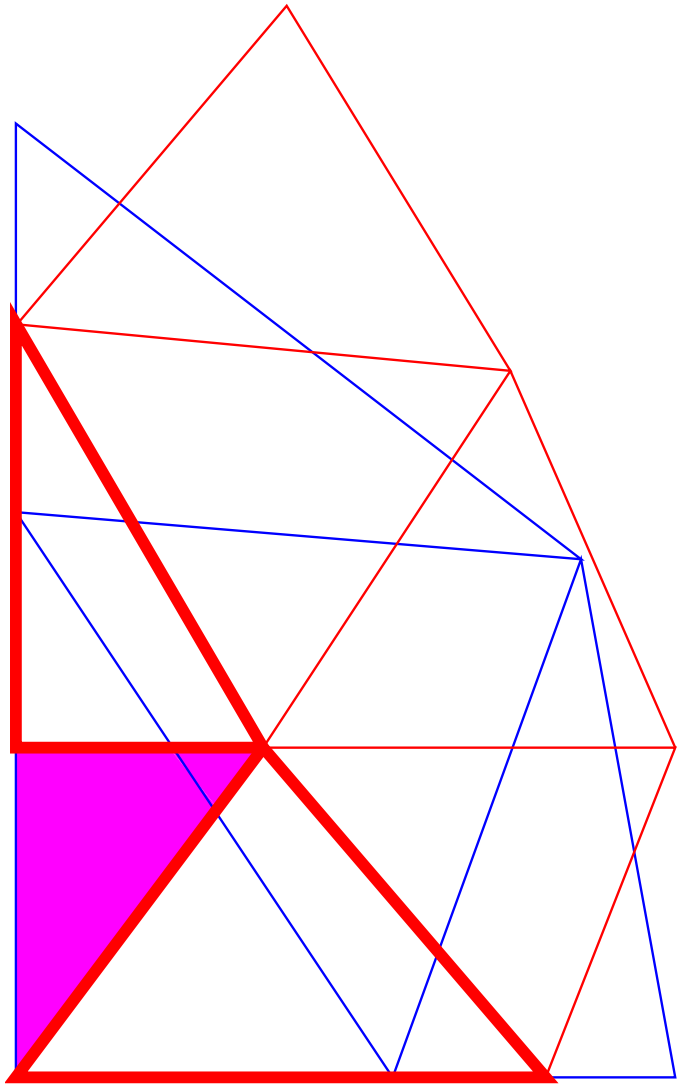
No Triangles Left to Intersect: red Triangle Finished

```
Ta and Tb two lists of simplices;
bl=simplex of Tb;
bil=simplex of Ta intersecting bl;
while length(bl)>0
  bc=bl(1); bl=bl(2:end);
  al=bil(1); bil=bil(2:end);
>   while length(al)>0
      ac=al(1);  al=al(2:end);
      [P,nc,Mc]=integrate(ac,bc);
      if ~isempty(P)
        M(ac,bc)=M(ac,bc)+Mc;
        al=[al neighbors(ac)];
        n(nc)=ac;
      end;
  end;
  bl=[bl untreatedneighbors(bc)];
  bil=[bil n(untreatedneighbors(bc))];
end;
```



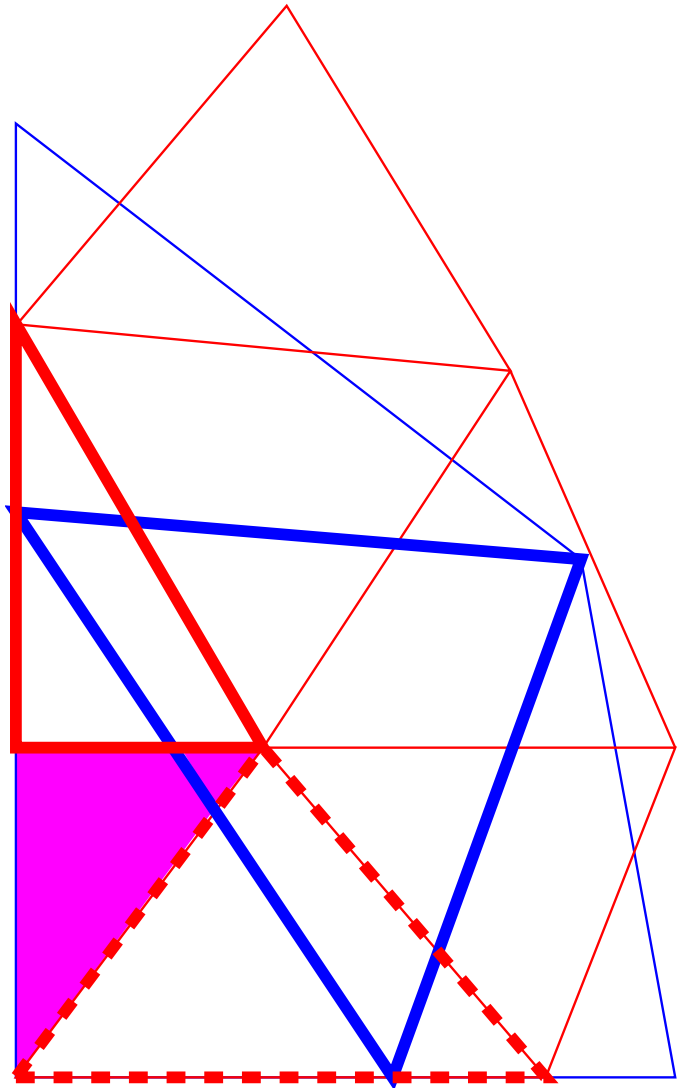
Add Neighbors of Red Triangle to be Treated

```
Ta and Tb two lists of simplices;
bl=simplex of Tb;
bil=simplex of Ta intersecting bl;
while length(bl)>0
  bc=bl(1); bl=bl(2:end);
  al=bil(1); bil=bil(2:end);
  while length(al)>0
    ac=al(1);  al=al(2:end);
    [P,nc,Mc]=integrate(ac,bc);
    if ~isempty(P)
      M(ac,bc)=M(ac,bc)+Mc;
      al=[al neighbors(ac)];
      n(nc)=ac;
    end;
  end;
>   bl=[bl untreatedneighbors(bc)];
>   bil=[bil n(untreatedneighbors(bc))];
end;
```



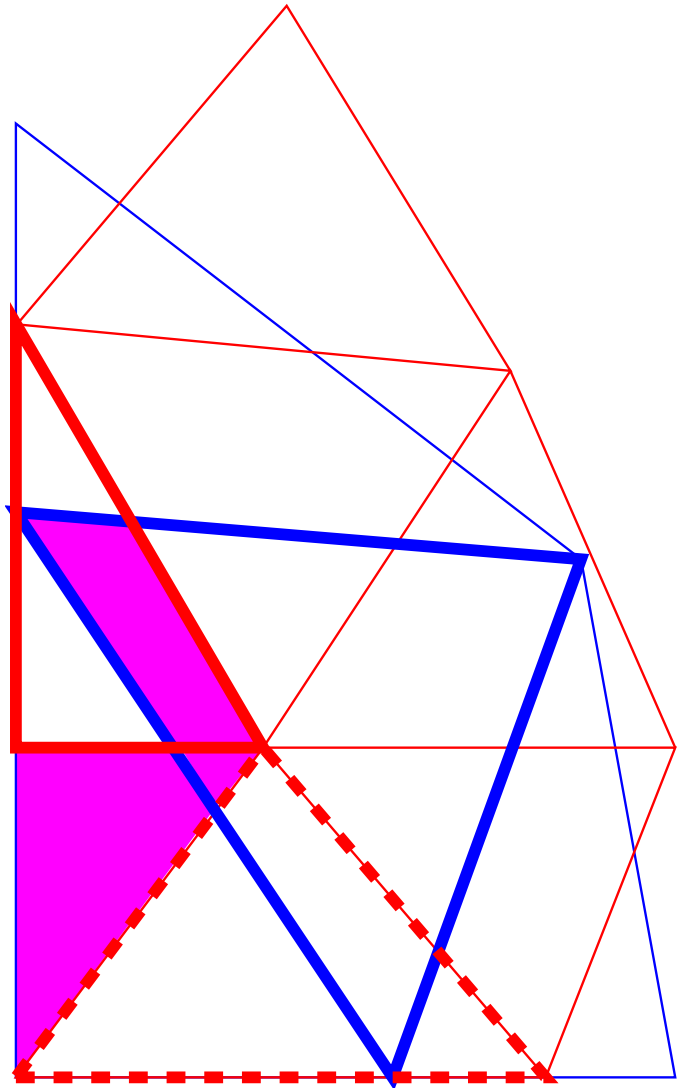
Pick First Red Triangle and Blue Starting Candidate

```
  Ta and Tb two lists of simplices;
  bl=simplex of Tb;
  bil=simplex of Ta intersecting bl;
  while length(bl)>0
>   bc=bl(1); bl=bl(2:end);
>    al=bil(1); bil=bil(2:end);
    while length(al)>0
      ac=al(1);  al=al(2:end);
      [P,nc,Mc]=integrate(ac,bc);
      if ~isempty(P)
        M(ac,bc)=M(ac,bc)+Mc;
        al=[al neighbors(ac)];
        n(nc)=ac;
      end;
    end;
    bl=[bl untreatedneighbors(bc)];
    bil=[bil n(untreatedneighbors(bc))];
  end;
```



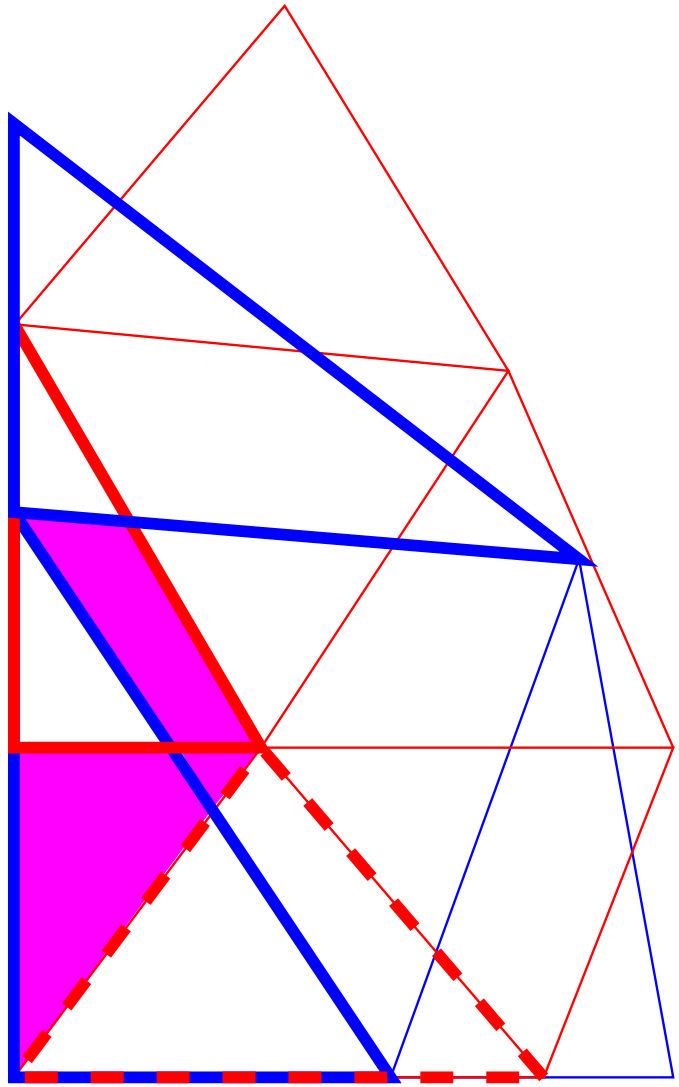
Compute Their Intersection

```
  Ta and Tb two lists of simplices;
  bl=simplex of Tb;
  bil=simplex of Ta intersecting bl;
  while length(bl)>0
    bc=bl(1); bl=bl(2:end);
    al=bil(1); bil=bil(2:end);
    while length(al)>0
      ac=al(1);  al=al(2:end);
>     [P,nc,Mc]=integrate(ac,bc);
      if ~isempty(P)
        M(ac,bc)=M(ac,bc)+Mc;
        al=[al neighbors(ac)];
        n(nc)=ac;
      end;
    end;
    bl=[bl untreatedneighbors(bc)];
    bil=[bil n(untreatedneighbors(bc))];
  end;
```



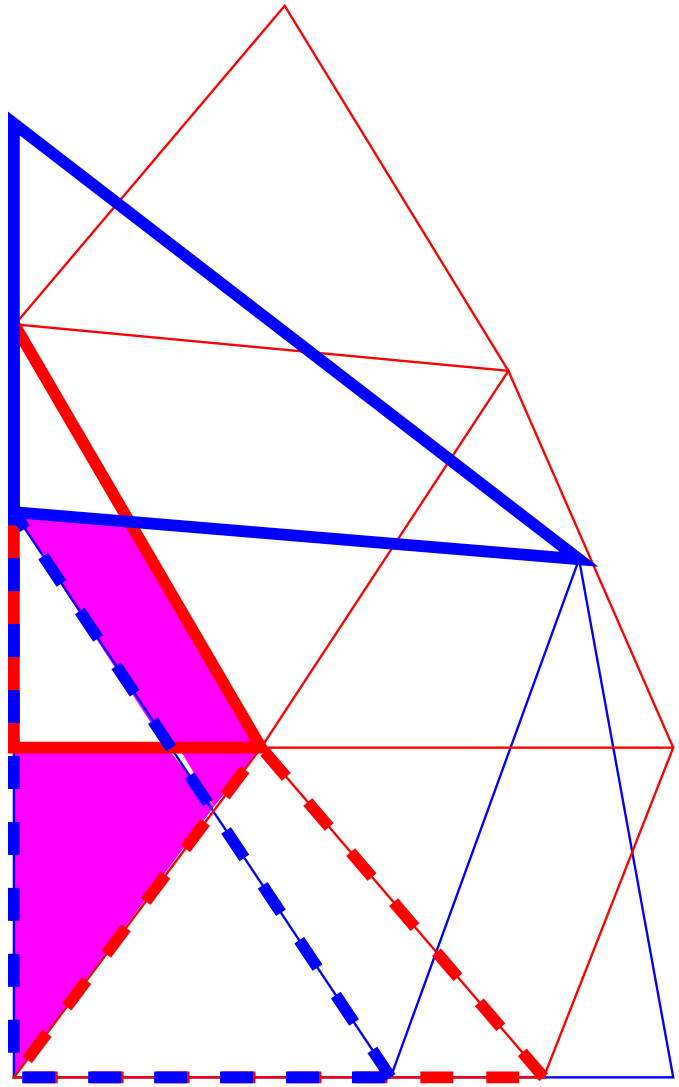
Add Blue Neighbors to Continue

```
Ta and Tb two lists of simplices;
bl=simplex of Tb;
bil=simplex of Ta intersecting bl;
while length(bl)>0
  bc=bl(1); bl=bl(2:end);
  al=bil(1); bil=bil(2:end);
  while length(al)>0
    ac=al(1);  al=al(2:end);
    [P,nc,Mc]=integrate(ac,bc);
    if ~isempty(P)
      M(ac,bc)=M(ac,bc)+Mc;
>     al=[al neighbors(ac)];
>     n(nc)=ac;
    end;
  end;
  bl=[bl untreatedneighbors(bc)];
  bil=[bil n(untreatedneighbors(bc))];
end;
```



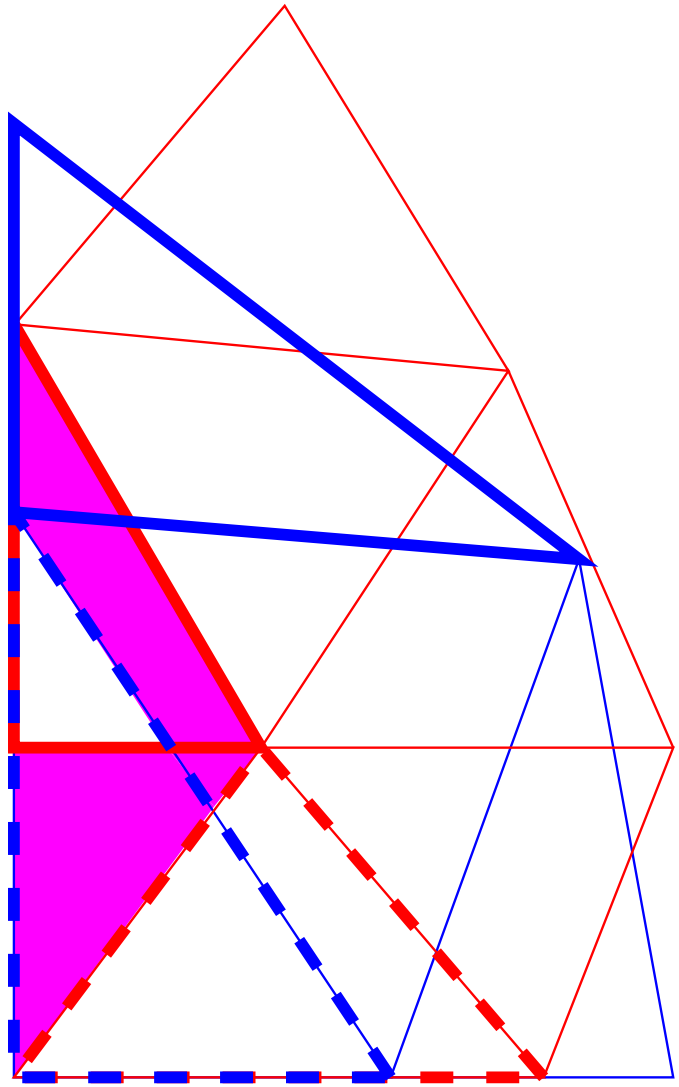
Pick First Blue Neighbor

```
  Ta and Tb two lists of simplices;
  bl=simplex of Tb;
  bil=simplex of Ta intersecting bl;
  while length(bl)>0
    bc=bl(1); bl=bl(2:end);
    al=bil(1); bil=bil(2:end);
    while length(al)>0
>     ac=al(1);  al=al(2:end);
      [P,nc,Mc]=integrate(ac,bc);
      if ~isempty(P)
        M(ac,bc)=M(ac,bc)+Mc;
        al=[al neighbors(ac)];
        n(nc)=ac;
      end;
    end;
    bl=[bl untreatedneighbors(bc)];
    bil=[bil n(untreatedneighbors(bc))];
  end;
```

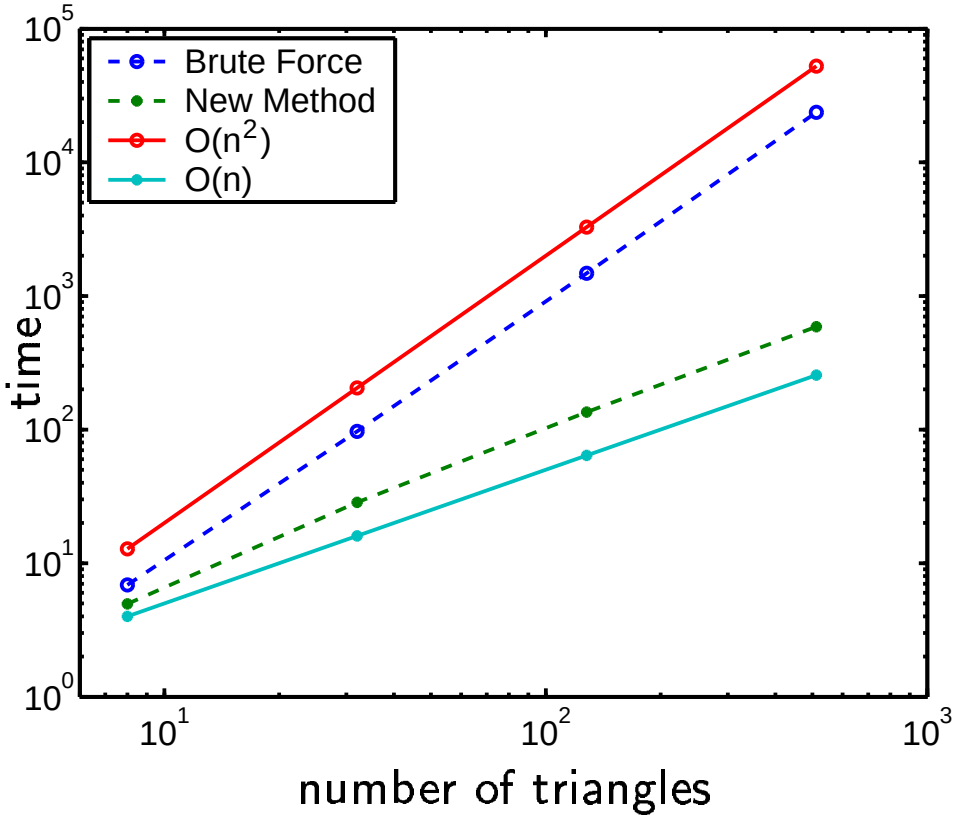


Compute Intersection and Continue Like This

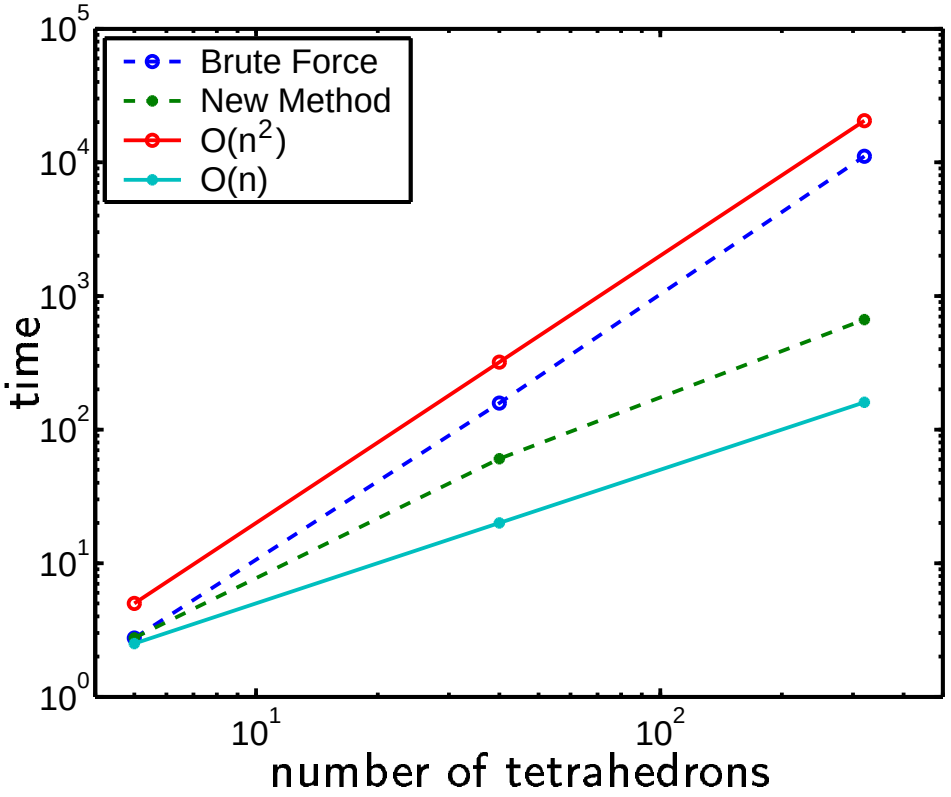
```
  Ta and Tb two lists of simplices;
  bl=simplex of Tb;
  bil=simplex of Ta intersecting bl;
  while length(bl)>0
    bc=bl(1); bl=bl(2:end);
    al=bil(1); bil=bil(2:end);
    while length(al)>0
      ac=al(1);  al=al(2:end);
>     [P,nc,Mc]=integrate(ac,bc);
      if ~isempty(P)
        M(ac,bc)=M(ac,bc)+Mc;
        al=[al neighbors(ac)];
        n(nc)=ac;
      end;
    end;
    bl=[bl untreatedneighbors(bc)];
    bil=[bil n(untreatedneighbors(bc))];
  end;
```



Complexity Measurements



twenty projections in 2d



5 projections in 3d

Conclusions

The projection matrix for non-matching grid projections can be computed with linear complexity, if neighboring information is available.

Further results and future work:

- Generalization to non-matching grids of interfaces in slightly different spatial locations $\Longrightarrow$ contact problems.
- Floating point arithmetic analysis of the intersection algorithm.

Algorithms can be downloaded at `www.unige.ch/~gander`