1 Changelog

1.1 The Original Algorithm

In fact, this section describes the basic structure of Coho. Because the original implementation does not have very clear architecture. I do not want to spend lots of time to explain the simple problems or updates. Therefore, I choose a simple but stable version to show the basic idea of Coho.

The main function of Coho is:

```
function ph = coho(ph)
  while(~done)
    ph = forward(ph) % compute forward projectagon
  end
```

The $forward$ function is:

```
function ph = forward(ph,bloat)
  %bloat is the maximum bloat amount allowed
  ph = ph_model(ph,bloat); % compute the model
  step = ph_maxTimeStep(ph,bloat); % compute maximum time step
  ph = ph_forward(ph,step);
```

The ph_model function is

```
function ph = ph_model(ph,bloat)
  bloatPh = bloat(ph,bloat)
  for each slice
    for each face
      bloatFace = bloat(bloat(face,-2*bloat),bloat); % 2*bloat inward, 1 bloat outward
      modelLP = intersect(bloatFace,bloatPh)
      model = model_create(modelLP); % compute linearize model
    end
  end
```

The $ph\_maxTimeStep$ function is

```
function step = ph_maxTimeStep(ph,bloat)
  for each slice
    for each face
      maxDot = lp_maxDot(model,modelLP) % find maximum derivative by lp
      step = min(step,bloat./maxDot);
    end
  end;
```

The $ph\_forward$ function is

```
function ph = ph_forward(ph,step)
  for each slice
```

```
    for each face
      bloatFace = bloat(face,-bloat); % 1 bloat inward
      bloatFace = intersect(bloatFace,ph);
      forwardLP = int_forward(bloatFace,model); % compute forward lp
      projs{i} = lp_project(forwardLP);
    end
    slice = poly_union(projs); % union of projected polygons
    slice = poly_simplify(slice); % reduce # of edges
  end
```

1.2 Current Implementation

The implemented algorithm is more complicated. There are several important changes.

1.2.1 Bloat Amount and Time Step

The computation of time step is too conservative and expensive. The tiny time step also increases the accumulated error and decreases performance. Therefore, a guess-verify algorithm is implemented to increase the time step. The algorithm bases on an assumption that *a time step is valid if the forward projectagon does not exceed the bloated projectagon.* The basic idea is that the algorithm guess a time step at the beginning and verify validity at the end. Because the projectagon changes a little for one step, the ¡bloat,step¿ pair from previous step is a good initial guess.

The updated $forward$ function is:

```
function [ph,bloat,step] = forward(ph,bloat,step)
  do{
    bloat = bloat*(min(maxBloat./bloat));
    step = step*(min(maxBloat./bloat)) %initial guess

    while(true) % find a valid <bloat, step> pair
      ph = ph_model(ph,bloat,step); % compute the model
      [newPh,ph] = ph_forward(ph,bloat,step); % forward it.
      realBloat = ph_realBloat(ph); %
      if( realBloat <= bloat )
        break;
      else % decrease time step
        step = step/2; update bloat;
      end
    end

    while(realBloat << bloat) % reduce error
      bloat = realBloat;
      ph = ph_model(ph,bloat,step);
```

```
      [newPh,ph] = ph_forward(ph,bloat,step);
      realBlot = ph_realBloat(ph);
    end
  }while(bloat << maxBloat)
  ph = ph_trim(newPh); % clip ph to make it feasible to the hull
```

The function guess a bloat and step based on the ones from previous steps until a valid step is found. Then it decrease the bloat to reduce modeling error as possible.

The validity of time step is checked by computing the forward distance of each face.

```
function realBloat = ph_realBloat(ph)
  for each slice
    for each face
      solve optimization problem
        min bloat
        s.t. bloated face contains the projection polygon
      realBloat = max(realBloat,bloat);
    end
  end
```

1.2.2 ph_model and ph_forward

Another important improvement is to reduce the approximation error.

The first simple idea is to use different bloat for different variables and directions. Because in a circuit, usually there are some signals changes much faster than others. The implementation is easy. Another update is that we use multiple methods to compute the linearize model, and use the intersection of the result to reduce linearization error.

In the original algorithm, we move forward a bloated face, which is usually too thick than necessary. In the current version, only the face is moving forward. Of course, the height of the face is increased for soundness. Correspondingly, the face is bloated by one instead of two bloat inward to compute the model.

Interval Closure method is applied to reduce error when the projectagon is not convex. Instead computing the intersection of face and the hull of projectagon, the interval closure method uses the concave polygons to reduce the height of face.

Finally, usually the real bloat is close to bloat for only a small amount of faces, the real bloat much smaller than bloat for others. Therefore, after the model is created, the maximum derivative is computed thus the maximum forward distance can be estimated given the time step. If the moving distance is much smaller than bloat, the model is recomputed with a much small bloat, which will reduce the modeling error a lot.

After these major changes, the *ph_model* function is

```
function ph = ph_model(ph,bloat,step)
```

```
  for each slice
    for each face
      while(true)
        bloatFace = bloat(face,-2*bloat);  % bloat inward by 2*bloat
        bbox = ph_interval(ph,bloatFace); % find the height of face by interval closure.
        heightLP = bloat(bbox,bloat); % increase the height of face for soundness
        modelLP = intersect(bloat(face,bloat), heightLP);
        models{i} = model_create(modelLP,methods(i)); % create model using several metho
        maxDot = lp_maxDot(model,modelLP);
        dist = maxDot*step;
        if(dist<<bloat)
          bloat = dist;
        else
          break;
        end;
      end;
    end
  end
```

The *ph_forward* function is

```
function ph = ph_forward(ph,bloat,step)
  for each slice
    for each face
      forwardFace = intersect(face,heightLP); % bound the height
      for each model
        forwardLP = int_forward(forwardFace,model);
        verts = lp_project(forwardLP);
        projs{i} = intersect(projs{i}, verts);
      end;
    end;
    slice = poly_union(projs);
    slice = poly_simplify(slice);
  end;
```

1.2.3 Feasibility

With smaller approximation error. Another problem appears: *a face might be infeasible to other slices*. This make the $forwardLP$ a infeasible problem and the *lp_project* function can not handle the case. Therefore, the projectagon should be clipped to remove the infeasible face at the end.

The *ph_trim* function is

```
function ph = ph_trim(ph)
  do{
    for each slice
      hull = lp_project(ph,slice); % project ph onto this hull
```

```
    poly = intersect(poly,hull); % clip the polygon by projected polygon
  end;
  newPh = ph_create(polys,hulls);
}while(newPh~=ph)
```

However, the face is only feasible to the convex hull of the projectagon. It is not a problem without interval closure. With interval closure method, infeasible face problem occurs again in the ph_model function, because an edge of a slice might not be feasible to polygons of other slices, thought it is feasible to convex hulls. We have not had an algorithm to clip the projectagon to make each face feasible to concave polygons of other slices.