Chapter 1

Linearization

This package computes the dynamic function and approximates it as an linear differential inclusion. The *Integrator* package integrate the LDI and compute the new reachable space. The input is the circuit and its current projectagon, the output is the LDI $\dot{x} \in Ax + b \pm err$.

The *model_vdot* function is the interface to other package. It compute the LDI for input clock, mos transistors and keeper circuit and then combine them.

The LDI for input and keeper is not difficult. It calls the mrg_fit function which compute the best linear fitting of two polynomial edges by linear programming. Because the number of points is small, the algorithm is efficient.

There are several methods to compute the linear fitting for mos transistors. Of course, we can use linear programming $linfit\_lp$. However, there are huge number of points (256x256x256), this method is usually not efficient.

We can use the linear least square method to find the "best" fitting which minimizes the L_2 error. However, the "regression" method in matlab is not efficient enough. Because the region is a 3D cube, we can optimize it as in lls (more detail later).

For each grid point, the $err = ids - c * [v; 1]$, therefore, err is convex with c. And convexity is maintained by max/min. Therefore, the linear fitting error is convex with c. Therefore, we can use local search method ($fmincon$) to find the optimal result. We implement it in $linfit\_ls$.

Future, for nmos(pmos) (not for nmos2), we know its ids is convex or concave with vs vg vd, we can use the convexity to find the optimal point which gives max/min error for a given c. Therefore, we can use nested $fmincon$ to solve the problem. It does not work for nmos2 device. However, during the implementation, there are too many numerical problems to use in practice.

Given a linear fitting c, we should compute the corresponding error. When the region is a cube, we have $linfitErr$ and $linfitErr\_conv$ functions. $linfitErr$ use brute force method to compute the error for all points and then find the max/min one. $linfitErr\_conv$ function takes advantage of the convexity of ids for nmos/pmos and search the extreme point more efficiently (not work for nmos2 device). However, generally $linfitErr$ is faster because it is highly

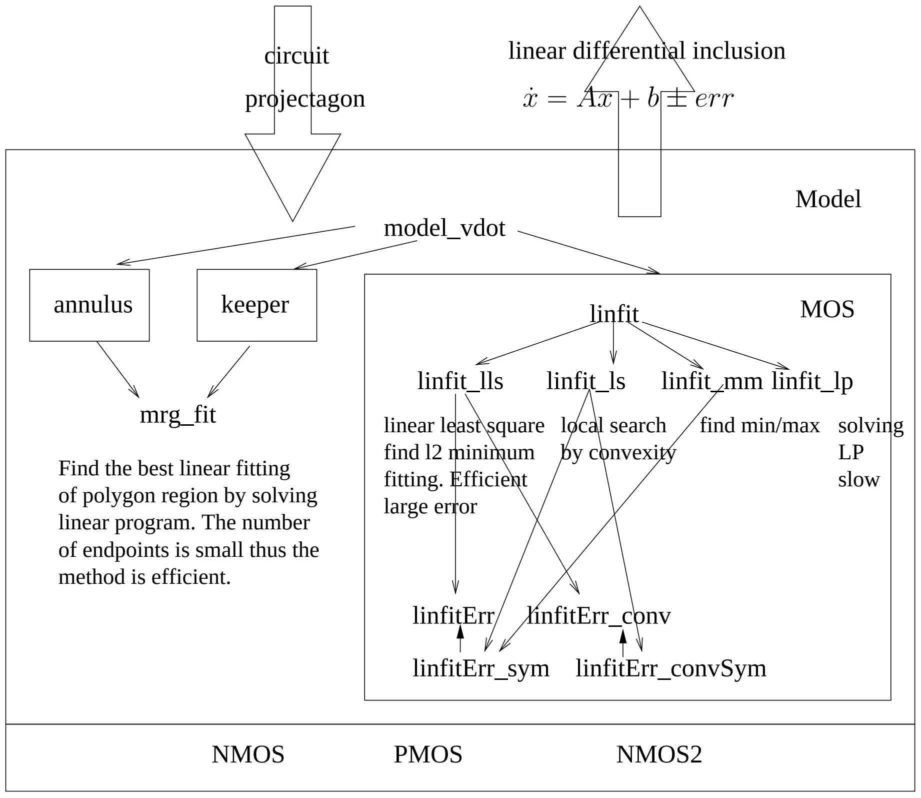

circuit
projectagon
linear differential inclusion
$\dot{x} = Ax + b \pm err$
Model
model_vdot
annulus
keeper
mrg_fit
Find the best linear fitting of polygon region by solving linear program. The number of endpoints is small thus the method is efficient.
MOS
linfit
linfit_lls
linfit_ls
linfit_mm
linfit_lp
linear least square find l2 minimum fitting. Efficient large error
local search by convexity
find min/max
solving LP slow
linfitErr
linfitErr_conv
linfitErr_sym
linfitErr_convSym
NMOS
PMOS
NMOS2

Figure 1.1: The architecture of Model package

vectorized. $linfitErr\_conv$ is better only when the cube is huge.

When the region is a cube and a lp. We use *pickup* (also used by $linfitErr$) function to find all grid points within the region. Then we can also use the brute force method as $linfitErr\_sym$. We also provide similar wrapper function $linfitErr_conv Sym$ to $linfit\_lp$ and other functions.

The ids data is from Hspice simulation. It is 3D. It has no other requirements. Only $linfitErr_conv$ function requires the convexity of the ids data.

1.1 Least Square Method

1.1.1 Problem

Given a m dimensional function $y = f(x_1, \cdots, x_m)$, what is the best least square fit?

The interface of the function is

```
function [b, ibnds, A, u] = lls(M, v)
```

The function f is discretized. For a grid point p, the function value is $y(p) = M.ids(p)$. $M.ids$ is a m dimensional table. For the point p, its d^{th} coordinate is

$$x_d(p) = v_d(ind_{dp}) = M.v0(d) + M.dv(d) \cdot ind_{dp} \tag{1.1}$$

where ind_p is the matrix index along d^{th} dimension of p. $M.YC$should be precomputed as

$$M.YC = cumsum(\cdots cumsum(M.ids, 1) \cdots m) \tag{1.2}$$

v is the bounding box of variables $x_1, \cdots x_m$.

1.1.2 Our Solution

First, we compute the hypercube

$$ibnds = \begin{bmatrix} l_1 & h_1 \\ \cdots & \cdots \\ l_m & h_m \end{bmatrix} \tag{1.3}$$

which contains the bounding box v.

Then we compute the size of the cube

$$n_d = h_d - l_d + 1 \tag{1.4}$$

$$n = \prod_{d=1}^{m} n_d \tag{1.5}$$

Let us assume the linear fit function is

$$yy(p) = b' * \begin{bmatrix} x_1(p) \\ \cdots \\ x_m(p) \\ 1 \end{bmatrix} = \sum_{d=1}^{m+1} b_d \cdot x_d(p) \tag{1.6}$$

Then the error is

$$err(b) = \sum_{p=1}^{n} (\ y(p) - yy(p)\ )^2 \tag{1.7}$$

To minimize the error, we compute its derivative

$$\begin{aligned}
\frac{\partial err}{\partial b_d} &= \frac{\partial}{\partial b_d}\sum_{p=1}^{n}(\ y(p) - yy(p)\ )^2 \\
&= \sum_{p=1}^{n}\frac{\partial}{\partial b_d}(\ y(p) - yy(p)\ )^2 \\
&= \sum_{p=1}^{n} 2\cdot(yy(p) - y(p))\cdot\frac{\partial}{\partial b_d}(yy(p) \\
&= \sum_{p=1}^{n} 2\cdot(yy(p) - y(p))\cdot x_d(p) \\
&= 2\cdot\sum_{p=1}^{n} x_d(p)\cdot yy(p) - 2\cdot\sum_{p=1}^{n} x_d(p)\cdot y(p) \\
&= 2\cdot\sum_{p=1}^{n} x_d(p)\cdot\sum_{dd=1}^{m+1} b_{dd}\cdot x_{dd}(p) - 2\cdot\sum_{p=1}^{n} x_d(p)\cdot y(p) \\
&= 2\cdot\sum_{dd=1}^{m+1} b_{dd}\cdot\sum_{p=1}^{n} x_d(p)\cdot x_{dd}(p) - 2\cdot\sum_{p=1}^{n} x_d(p)\cdot y(p) \\
&= 2\cdot\sum_{dd=1}^{m+1} b_{dd}\cdot A - 2\cdot u
\end{aligned}$$

A, u are defined as:

$$\begin{aligned}
A(d_1, d_2) &= \sum_{p=1}^{n} x_{d_1}(p)\cdot x_{d_2}(p) \\
u(d) &= \sum_{p=1}^{n} x_d(p)\cdot y(p)
\end{aligned}$$

err is minimized when $\frac{\partial err}{\partial b_d} = 0$. This means

$$b = A \setminus u. \tag{1.8}$$

1.1.3 How to compute u vector

u is a $m+1$ vector. The d^{th} element is the sum of product of y and x_d over all points in the cube.

$$\begin{aligned} u(d) &= \sum_{p=1}^{n} x_d(p) \cdot y(p) \\ &= \sum_{ind=l_d}^{h_d} v_d(ind) Y_d(ind) \end{aligned}$$

where $Y(ind)$ is the sum of all y values on the face $x_d = v_d(ind)$ (the ind^{th} slice on the d^{th} dimension.

$$\begin{aligned} Y_d(ind) &= \sum_{p \in \{x_d(p) = v_d(ind)\}} y(p) && (1.9) \\ v_d(ind) &= M.v0(d) + M.dv(d) * ind && (1.10) \end{aligned}$$

we define another function YCV_d which is the sum of all points on the left of face $x_d = v_d(ind)$.

$$\begin{aligned} YCV_d(ind) &= \sum_{i=1}^{ind} Y_d(i) && (1.11) \\ &= \sum_{p \in \{x_d(p) \le v_d(ind)\}} y(p) && (1.12) \\ YCV_d(0) &= 0 && (1.13) \\ Y_d(ind) &= YCV_d(ind) - YCV_d(ind-1) && (1.14) \end{aligned}$$

Therefore,

$$\begin{aligned} u(d) &= \sum_{ind=l_d}^{h_d} v_d(ind)(YCV_d(ind) - YCV_d(ind-1)) && (1.15) \\ &= v_d(h_d) \cdot YCV_d(h_d) + (v_d(h_d) - v_d(h_d - 1)) \cdot YCV_d(h_d - 1) + \cdots \\ &\quad + (v_d(l_d + 1) - v_d(l_d)) \cdot YCV_d(l_d) - v_d(l_d) \cdot YCV_d(l_d - 1) \\ &= v_d(h_d) \cdot YCV_d(h_d) - M.dv(d) \cdot \sum_{ind=l_d}^{h_d - 1} YCV_d(ind) - v_d(l_d) \cdot YCV_d(l_d - 1) \end{aligned}$$

The special case is $d = m+1$, $v_d(:) = 1$, $u(m+1)$ is the sum of all of the values in the cube. The equation is simplified as

$$u(m+1) = YCV_d(h_d) - YCV_d(l_d - 1) \qquad (1.16)$$

for any other dimension d.

How to compute YCV_d vector

To compute $u(d)$, $YCV_d(ind), l_d - 1 \leq ind \leq h_d$ should be computed. It is implemented in *ycvHelp* function.

$YCV_d(ind)$ is the sum of all points in the cube *ibnds*. We have precomputed $M.YC = cumsum(cumsum(...cumsum(M.ids, 1)..., d-1), d)$. Therefore $M.YC(p)$ is the sum of all point in the cube between $(1, 1..., 1)$ and p. We should find all points which are the corners of the hypercube *ibnds*. Then $YCV_d(ind)$ is computed by adding or subtracting the value at each corner to the final sum. The corner that corresponds to the maximizing all variables (the top corner) is added. Other corners are added if a path from the top corner to the current corner has even length, and subtracted if the length of the path is odd.

Take a 2D example, see Figure 1.2.

$$\begin{aligned} YCV_3(i_3) &= \sum_{ind=1}^{i_3} Y_d(ind) \\ &= M.YC(p1) - M.YC(p2) - M.YC(p4) + M.YC(p3) \end{aligned}$$

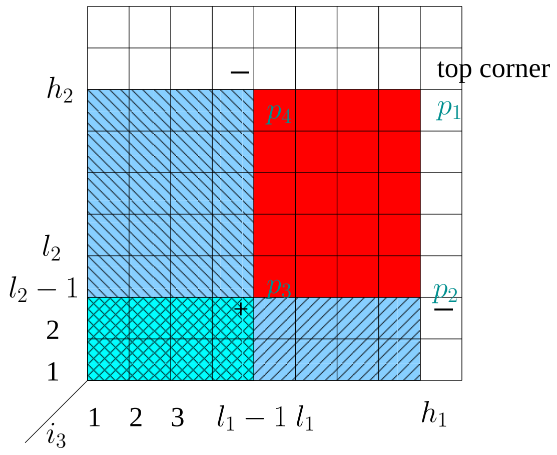


Figure 1.2: The computation of YCV

Therefore, the code is

```
% M.YC = cumsum(...cumsum(M.ids,1)...m);
% ibnds is the hypercube, ibnds(i,1)>=1
% j: YCV_j
% i: i = cell(d,1); i{j} = ibnds(j,1)-1:ibnds(j,2)
%    Compute YCV_j(ibnds(j,1)-1) to YCV_j(ibnds(j,2)).
% nv: the number of dimensions, m.
% k:  the dimension working on. 1 as initial.
```

```
function ycv = ycvHelp(YC, ibnds, i, j, nv, k)
if(k == j)
    ycv = ycvHelp(YC, ibnds, i, j, nv, k+1);
elseif(k <= nv)
    i{k} = ibnds(k,1)-1; % lower corner
    if(i{k} == 0) % the boundary
        ycv1 = zeros(size(i{j}))';
    else
        ycv1 = ycvHelp(YC, ibnds, i, j, nv, k+1);
    end;
    i{k} = ibnds(k,2); % upper corner
    ycv2 = ycvHelp(YC, ibnds, i, j, nv, k+1);
    ycv = ycv2 - ycv1;
else
    ycv = reshape(YC(i{:}), [], 1);%corners for YCV(ind) from ld-1 to hd.
end;
```

However, when $ibnds(j,1) == 1$, we use $i\{j\} = 1 : ibnds(j,2)$ to make this function work. The returned vector is for $l_d : h_d$ rather than $l_d - 1 : h_d$ as expected. We should insert 0 in to ycv vector. The bug of the previous version is that we use $ycv(l_d)$ instead of $ycv(l_d - 1)$ to compute $u(d)$.

Therefore, the code to compute u vector is:

```
u = zeros(nv+1, 1);
for j = 1:nv
i = cell(nv, 1);
    if(ibnds(j,1)>1)
        i{j} = (ibnds(j,1)-1) : ibnds(j,2);
        ycv = ycvHelp(M.YC,ibnds,i,j,nv,1);
    else
        i{j} = ibnds(j,1):ibnds(j,2);
        ycv = ycvHelp(M.YC,ibnds,i,j,nv,1);
        ycv = [0;ycv]; % insert 0 for ycv(ibnds(j,1)-1);
    end;
    u(j) = vbnds(j,2)*ycv(end) - M.dv(j)*sum(ycv(2:end-1)) - vbnds(j,1)*ycv(1);
    if(j==nv)
        u(nv+1) = ycv(end) - ycv(1);
    end;
end;
```

1.1.4 How to compute A matrix

We know

$$A(d_1, d_2) = \sum_{p=1}^{n} x_{d_1}(p) \cdot x_{d_2}(p)$$

The diagonal elements of A are given by

$$
\begin{aligned}
A(d,d) &= \sum_{p=1}^{n} x_d(p)\cdot x_d(p) \\
&= \sum_{ind=l_d}^{h_d} v_d(ind)^2 \cdot \frac{n}{n_d} \\
&= \sum_{ind=l_d}^{h_d} (M.v0(d) + M.dv(d)\cdot ind)^2 \cdot \frac{n}{n_d} \\
&= \sum_{ind=l_d}^{h_d} (M.v0(d)^2 + 2\cdot M.v0(d)\cdot M.dv(d)\cdot ind + M.dv(d)^2\cdot ind^2)\cdot \frac{n}{n_d} \\
&= (n_d\cdot M.v0(d)^2 + 2\cdot M.v0(d)\cdot M.dv(d)\sum_{ind=l_d}^{h_d} ind + M.dv(d)^2\cdot \sum_{ind=l_d}^{h_d} ind^2)\cdot \frac{n}{n_d} \\
&= \left(n_d\cdot M.v0(d)^2 + 2\cdot M.v0(d)\cdot M.dv(d)\cdot \frac{(l_d+h_d)\cdot n_d}{2} + M.dv(d)^2\cdot \frac{2(h_d^3-l_d^3)+3(h_d^2+l_d^2)+(h_d-l_d)}{6}\right)\cdot \frac{n}{n_d} \\
&= n\cdot\left(M.v0(d)^2 + M.v0(d)\cdot M.dv(d)\cdot (l_d+h_d) + M.dv(d)^2\cdot \frac{2(h_d^3-l_d^3)+3(h_d^2+l_d^2)+(h_d-l_d)}{6\cdot n_d}\right)
\end{aligned}
$$

The non-diagonal elements of A are given by

$$
\begin{aligned}
A(d_1,d_2) &= \frac{n}{n_{d_1}\cdot n_{d_2}} \sum_{ind_1=l_{d_1}}^{h_{d_1}} \sum_{ind_2=l_{d_2}}^{h_{d_2}} v_{d_1}(ind_1)\cdot v_{d_2}(ind_2) \\
&= \frac{n}{n_{d_1}\cdot n_{d_2}} \sum_{ind_1=l_{d_1}}^{h_{d_1}} v_{d_1}(ind_1)\cdot \sum_{ind_2=l_{d_2}}^{h_{d_2}} v_{d_2}(ind_2) \\
&= \frac{n}{n_{d_1}\cdot n_{d_2}}\cdot n_{d_1}\frac{v_{d_1}(l_{d_1})+v_{d_1}(h_{d_1})}{2}\cdot n_{d_2}\frac{v_{d_2}(l_{d_2})+v_{d_2}(h_{d_2})}{2} \\
&= n\cdot \frac{v_{d_1}(l_{d_1})+v_{d_1}(h_{d_1})}{2}\cdot \frac{v_{d_2}(l_{d_2})+v_{d_2}(h_{d_2})}{2}
\end{aligned}
$$

Therefore, the code is

```
midv = sum(vbnds,2)/2;
A = midv * midv';
n   = n(1:end-1);
mid = sum(ibnds,2)/2;
nn  = size(A,1);
dd  =    M.v0 .* (M.v0 + (2 .* M.dv .* mid)) ...
    + M.dv.^2 .* (diff(2*ibnds.^3 + ibnds, 1, 2) + 3*sum(ibnds.^2, 2)) ...
    ./ (6*n);
A(1:(nn+1):nn^2) = [dd', 1];
```

1.2 Quadratic Interplation Method

1.2.1 Quadratic Polynomial Approximation

Quadratic Polynomial

First, we can approximate a N dimensional function by a *quadratic polynomial* using N variables:

$$\begin{aligned}
f(x_1,\cdots,x_n) &= \begin{array}{ccccccccc} b_{1,1}x_1^2 & + & b_{1,2}x_1x_2 & + & \cdots & + & b_{1,n}x_1x_n & + & b_{1,n+1}x_1 \\ & + & b_{2,2}x_2^2 & + & \cdots & + & b_{2,n}x_2x_n & + & b_{2,n+1}x_2 \\ & & & & \cdots & & \cdots & & \cdots \\ & & & & & + & b_{n,n}x_n^2 & + & b_{n,n+1}x_n \\ & & & & & & & + & b_{n+1,n+1} \end{array} \\
&= \sum_{i=1}^{n+1}\sum_{j=i}^{n+1} b_{i,j}\cdot x_i\cdot x_j \qquad (1.17) \\
&\equiv u'\cdot B\cdot u
\end{aligned}$$

where matrix B is an upper triangular matrix and x_{n+1} is the constant variable.

Replacing coefficient b with a as

$$a_{i,j} = \begin{cases} b_{i,j} & i = j \\ \frac{b_{i,j}}{2} & i \neq j \end{cases}$$

the function can be rewritten as matrix form:

$$\begin{aligned}
f(x_1,\cdots,x_n) &= \begin{array}{ccccccccc} a_{1,1}x_1x_1 & + & a_{1,2}x_1x_2 & + & \cdots & + & a_{1,n}x_1x_n & + & a_{1,n+1}x_1 \\ a_{2,1}x_2x_1 & + & a_{2,2}x_2x_2 & + & \cdots & + & a_{2,n}x_2x_n & + & a_{2,n+1}x_2 \\ \cdots & & \cdots & & \cdots & & \cdots & & \cdots \\ a_{n,1}x_nx_1 & + & a_{n,2}x_nx_2 & + & \cdots & + & a_{n,n}x_nx_n & + & a_{n,n+1}x_n \\ a_{n+1,1}x_n & + & a_{n+1,2}x_n & + & \cdots & + & a_{n+1,n}x_n & + & a_{n+1,n+1} \end{array} \\
&= \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \\ 1 \end{bmatrix}' \cdot \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & a_{1,n+1} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & a_{2,n+1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} & a_{n,n+1} \\ a_{n+1,1} & a_{n+1,2} & \cdots & a_{n+1,n} & a_{n+1,n+1} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \\ 1 \end{bmatrix} \\
&\equiv u'\cdot A\cdot u \qquad (1.18) \\
&= sum(sum(A\cdot *(u\cdot u')))
\end{aligned}$$

As see above, there are several ways to represents a quadratic polynomial. In the following, we use A to represent the symmetric matrix, use B to represent the upper triangular matrix. However, it is lower triangular matrix in matlab code because it is column first in matlab. We use $\vec{B}$ to represent the vector who stores the upper (lower in matlab) triangular elements. Sometimes, the full version of B is need, we use $\widehat{B}$ to represent it. The relationship between them are

$$\begin{aligned} b &= B(tril(true(n+1,n+1))) \\ \widehat{B} &= A + A' - diag(A) \\ A &= (B+B')/2 \\ B &= tril(\widehat{B}) \end{aligned}$$

The convertion is implemented as matlab code

```
function mm = quadConvert(m,src,dst)
% mm = quadConvert(m,src,dst)
%  The function converts between quadratic matrix
%  'b': the coefficient of quadratic polynomial as a vector
%  'B': the matrix form of 'b', which is a lower triangular matrix
%  'Bfull': copy lower triangular to upper triangular except the diagnoal elements
%  'A": the symmetric matrix form of quadratic polynomial
%
% m: each column is one matrix to convert
% src,dst: matrix form
% mm: the result

[k,nc] = size(m);
% conver to A matrix first
switch(lower(src))
case lower('b')
    b = m;
    n = round((sqrt(8*k+1)-3)/2);
    ind = tril(true(n+1,n+1));
    map = zeros(n+1,n+1);
    map(ind) = 1:k;
    map = map+map'-diag(diag(map));
    Bfull = b(map,:);
    A = repmat(reshape((1+eye(n+1))/2,[],1),1,nc).*Bfull;
case lower('B')
    B = m;
    n = round(sqrt(k)-1);
    mB = reshape(B,n+1,[]);
    tB = reshap(mB',n+1,[]);
    Bfull = (mB+tB)/2;
    Bfull = reshape(Bfull,[],nc);
    A = repmat(reshape((1+eye(n+1))/2,[],1),1,nc).*Bfull;
case lower('Bfull')
    Bfull = m;
    n = round(sqrt(k)-1);
    A = repmat(reshape((1+eye(n+1))/2,[],1),1,nc).*Bfull;
case lower('A')
    A = m;
    n = round(sqrt(k)-1);
otherwise
    error('do not support now');
end

% convert A to dst
switch(lower(dst))
case lower('b')
    Bfull = reshape(reshape((2-eye(n+1)),[],1),1,nc).*A;
    ind = tril(true(n+1,n+1));
    b = Bfull(ind,:);
    mm = b;
case lower('B')
    Bfull = reshape(reshape((2-eye(n+1)),[],1),1,nc).*A;
    B = zeros(k,nc);
    ind = tril(true(n+1,n+1));
    B(ind,:) = Bfull(ind,:);
    mm = B;
case lower('Bfull')
    Bfull = reshape(reshape((2-eye(n+1)),[],1),1,nc).*A;
    mm = Bfull;
case lower('A')
    % done
    mm  = A;
otherwise
    error('do not support now');
end;
```

Interpolation[1]

In Coho, the ids table is usually huge[2]. Therefore, it uses a lot of memory and does not work for devices with more than 3 terminals. Interplated function can solve this problem. The idea is that given a N dimensional ids table, we split the whole region into small cubes or hyper-rectangles, and then approximate the ids function as quadratic polynomials for each cube. The number of cubes is much smaller than the size of original table. And only several parameter needs to saved for each cube. This also make it possible to run the program in the Nvidia video card in parallel.

Figure 1.3 and figure 1.4 shows the 2D and 3D examples of the idea. For the 2D case, the whole region is splitted into squares with length of L. For each intersection point p, we approximate the ids function by quadratic polynomial in a bigger square with length of $2L$. Therefore, any point is covered by four different squares. For example, the red region is covered by the gold, blue, green and magenta squares. For the 3D case, any point p in the blue cube is covered by eight different squares. For general case, each point is covered by 2^d different $d-dimensional$ cubes.

In each cube, the polynomial coefficient is computed using least square method. For all sampled points, $[x_1^2, x_1x_2, \cdots, x_1x_n, x_1, x_2^2, \cdots, x_2x_n, x_2, \cdots, x_n^2, x_n, 1]$ is computed as the input of least square method. The result of least square method is $\vec{B}$. For example, the parameter for 3D case is $\vec{B} = [b_1, \cdots, b_{10}]$ and the approximation function is:

$$
f(x_1,x_2,x_3) \quad = \quad \begin{array}{ccccccc}
b_1x_1^2 & + & b_2x_1x_2 & + & b_3x_1x_3 & + & b_4x_1 \\
 & + & b_5x_2^2 & + & b_6x_2x_3 & + & b_7x_2 \\
 & & & + & b_8x3^2 & + & b_9x_3 \\
 & & & & & + & b_{10}
\end{array}
$$

Give a point p, its coordinate x is first nomarlized according to the grid size as $\frac{x_i}{L_i}$. Then the relative position to the cube center is used as the polynomial variable. The relationship between the *normalized relative coordinate* Δx and *original coordinate* or *coordinate* x is shown in equation 1.19.

$$\Delta x_i = \frac{x_i}{L_i} - x0_i \tag{1.19}$$

[1]March 6, 2009

[2]For example, for a transistor, we have a 256x256x256 table

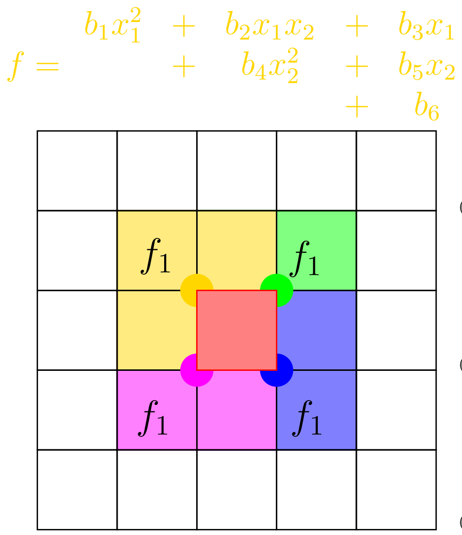

$f = b_1x_1^2 + b_2x_1x_2 + b_3x_1 + b_4x_2^2 + b_5x_2 + b_6$
f_1
f_1
f_1
f_1

Figure 1.3: Interplation Method (2D)

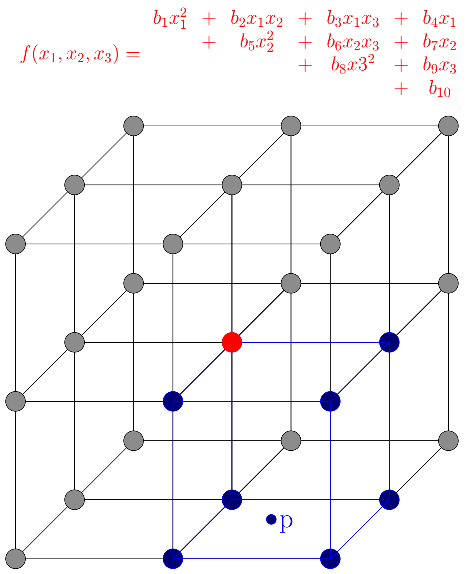

$f(x_1,x_2,x_3) = b_1x_1^2 + b_2x_1x_2 + b_3x_1x_3 + b_4x_1 + b_5x_2^2 + b_6x_2x_3 + b_7x_2 + b_8x3^2 + b_9x_3 + b_{10}$
p

Figure 1.4: Interplation Method (3D)

where x0 is the position of the center of the cube, it might be $\lceil \frac{x}{L} \rceil$ or $\lfloor \frac{x}{L} \rfloor$ depends on which cubes is working in. Obviously, the range for Δx_i is from -1 to 1. However, we will use x directly for the quadratic polynominal in the following section if the original coordinate is not presented.

Given a point, it is covered by 2^d different cubes thus has 2^d different approximation values. To make hte interpolation function smooth, we use an cosine window as the weight of each cube. The cosine window is defined as

$$w(x) = \sum_{i=0}^{n} a_i cos(i \cdot x \cdot \pi)$$

which should satisfies

$$\begin{aligned} w(0) &= 1 \\ w(x) + w(x-1) &= 1 \\ \frac{d^k f}{dx^k}(1) &= 0, \forall k \in [0, d] \end{aligned}$$

The second constraint make sure the sum of weight is always 1. Which requires the even coeffcient to be zero

$$\begin{aligned} a_0 &= \frac{1}{2} \\ a_{2k} &= 0, \forall k = [1, 2, \ldots]. \end{aligned}$$

The last constraint is used to make the interplation function C^d continous. As we know

$$\begin{aligned}
\frac{d^k f}{dx^k}(1) &= \sum_{i=0}^{n} i^k \cdot (-1)^{\lfloor k/2 \rfloor + i} \cdot a_i \\
&= \sum_{i \in [0,1:2:n]} i^k \cdot (-1)^{\lfloor k/2 \rfloor + i} \cdot a_i \\
\frac{d^k f}{dx^k}(1) &= 0, \forall k \in [1,3,5,\ldots]
\end{aligned}$$

a can be solved[3].

`ODE45` requires 4th order derivative to be continous. Thus we use

$$w(x) = \frac{1}{2} + \frac{75}{128}\cos(x\pi) - \frac{25}{256}\cos(3x\pi) + \frac{3}{256}\cos(5x\pi) \tag{1.20}$$

which make the function C^5 continous.

Thus, the interlation function is

$$f(x) = \sum_{i=1}^{2^n} w(\Delta x_1^i) \cdots w(\Delta x_n^i) f_i(\Delta x_1^i, \cdots, \Delta x_n^i) \tag{1.21}$$

For example, the interpolation equation for the 3D case is

$$\hat{f}(x_1, x_2, x_3) = \sum_{i=1}^{8} \begin{array}{c}
\left(\frac{1}{2} + \frac{75}{128}\cos(\Delta x_1^i \pi) - \frac{25}{256}\cos(3\Delta x_1^i \pi) + \frac{3}{256}\cos(5\Delta x_1^i \pi)\right) \cdot \\
\left(\frac{1}{2} + \frac{75}{128}\cos(\Delta x_2^i \pi) - \frac{25}{256}\cos(3\Delta x_2^i \pi) + \frac{3}{256}\cos(5\Delta x_2^i \pi)\right) \cdot \\
\left(\frac{1}{2} + \frac{75}{128}\cos(\Delta x_3^i \pi) - \frac{25}{256}\cos(3\Delta x_3^i \pi) + \frac{3}{256}\cos(5\Delta x_3^i \pi)\right) \cdot \\
f_i(\Delta x_1^i, \Delta x_2^i, \Delta x_3^i)
\end{array}$$

Therefore, the interpolation algorithm is

1. Compute the parameters of quadratic polynomial function using least square methods.

2. Given the position x of a point p, find the center of all cubes that cover the point by computing $\lceil \frac{x_i}{L_i} \rceil$, $\lfloor \frac{x_i}{L_i} \rfloor$.

3. For each cube, compute the normalized relative position of p by computing Δx.

4. Use the cosine window as the weight to interplate the approximated value from 2^d cubes.

The implementation is

[3] a can be computed by `cwin.m` function

```
function v = cosInterp(model,p)
% find all cubes that contains the points
[bs,xs] = findCubes(model,reshape(p,[],1));
nc = length(bs);
vv = zeros(nc,1);
for i=1:nc
    b = bs{i}; x = xs(:,i);
    xpi = x*pi;
    vv(i) = prod(1+cos(xpi)-1/9*cos(3*xpi))*quadPolyEval(b,x);
end;
v = sum(vv)/nc;

function v = quadPolyEval(b,x)
[n,np] = size(x);
u = [x;ones(1,np)];
dv = repmat(u,n+1,1).*reshape(repmat(reshape(u,[],1),1,n+1)',(n+1)^2,[]); % u*u'

if(numel(b)==(n+1)^2) % matrix A
    b = reshape(b,[],1);
else % b vector
    qpind = tril(true(n+1,n+1));
    dv = dv(qpind,:);
end;
v = sum(repmat(b,1,np).*dv,1);

function [bs,dv] = findCubes(model,p)
n = length(model.dv);
pos = (reshape(p,[],1)-model.v0)./model.dv; % grid position

% find all cubes that contain the point
bbox = [ max(1,min(model.nv-1,floor(pos))), max(2,min(model.nv,floor(pos)+1)) ];
ind = cell(n,1);
for i=1:n
    ind(i) = {bbox(i,1):bbox(i,2)};
end;
bs = reshape(model.data(ind{:}),[],1);

% compute relative position in each cube
dd = repmat(pos,1,2)-bbox;
choice = 2; np = choice^n;
ind = mod(floor( repmat(0:(np-1),n,1)./repmat(choice.^(0:(n-1))',1,np)), choice);
ind = sub2ind([n,choice],repmat((1:n)',1,np),ind+1);
dv = dd(ind);
```

Obviously, the normalized realtive coordinate of the same point p the the 2^n combinations of $[\frac{x}{L} - \lfloor\frac{x}{L}\rfloor]$. For example, the cooridinate is for $[(\Delta x, \Delta y), (\Delta x - 1, \Delta y), (\Delta x - 1, \Delta y - 1), (\Delta x, \Delta y - 1)]$ 2D case.Thus, it is easy to prove

$$\sum_{i=1}^{2^n} w(x_1)\cdots w(x_n) \quad = \quad 1$$

The derivative of quadratic polynomial function is

$$\frac{d(u * A * u')}{du} \quad = \quad 2Au \tag{1.22}$$

and the derivative of cosine window is

$$\frac{d(w(x))}{dx} \quad = \quad \left(-\frac{75}{128}\sin(x\pi) + \frac{75}{256}\sin(3x\pi) - \frac{15}{256}\cos(5x\pi)\right)\pi \tag{1.23}$$

Therefore, the derivative of interpolation function is

$$
\begin{aligned}
\frac{d(\hat{f}(x))}{dx_k} &= \sum_{i=1}^{2^n} \left(\begin{array}{l} w(x_1^i)\cdots w(\Delta x_{k-1}^i)\frac{d(w(\Delta x_k^i))}{dx_k} w(\Delta x_{k+1}^i)\cdots w(\Delta x_n^i)\cdot f(\Delta x^i) \\ +w(\Delta x_1^i)\cdots w(\Delta x_n^i)\cdot \frac{d(f(\Delta x^i))}{dx_k} \end{array} \right) \\
&= \frac{1}{L_i}\sum_{i=1}^{2^n} \left(\begin{array}{l} w(x_1^i)\cdots w(\Delta x_{k-1}^i)\frac{d(w(\Delta x_k^i))}{d\Delta x_k^i} w(\Delta x_{k+1}^i)\cdots w(\Delta x_n^i)\cdot f(\Delta x^i) \\ +w(\Delta x_1^i)\cdots w(\Delta x_n^i)\cdot \frac{d(f(\Delta x^i))}{d\Delta x_k^i} \end{array} \right) \\
&= \frac{1}{L_i}\sum_{i=1}^{2^n} \left(\prod_{j=1}^{n} w(\Delta x_j^i)\cdot \left(\frac{d(w(\Delta x_k^i))}{d\Delta x_k^i}\frac{1}{w(\Delta x_k^i)}\cdot f(\Delta x^i) + \frac{d(f(\Delta x^i))}{d\Delta x_k^i} \right) \right) \quad (1.24)
\end{aligned}
$$

The code is

```
function par = cosInterpPar(model,p)
n = length(model.dv);
% find all cubes that contains the points
[bs,xs] = findCubes(model,reshape(p,[],1));
nc = length(bs);
pars = zeros(n,nc);
for i=1:nc
    b = reshape(bs{i},[],1); x = xs(:,i);
    A = reshape(quadConvert(b,'b','A'),n+1,n+1);

    xpi = x*pi;
    cf = 1+cos(xpi)-1/9*cos(3*xpi);
    parcf = -(4*pi/3)*sin(xpi).^3;
    f = quadPolyEval(A,x);
    parf = 2*A*[x;1];

    pars(:,i) = prod(cf).*(f.*parcf./cf + parf(1:n,:));

end;

par = sum(pars,2)./(nc*model.dv);
```

`The interface is updated on April 2nd, 2009 to support n-d and vectorized voltage inputs.`

1.2.2 Linearization Method

Coho can only integrate linear ODE now. Therefore, the quadratic model needs to be linearized. Coho requires an interface like

```
function [b,err] = quadLinfit(model,bbox)
```

which computes the coefficient and error bound such that $f(x) \in b' * [x;1] \pm err$.

First, to make the problem simple, we ignore the cosine window. That is, the algorithm is changed to

1. Compute the parameters of quadratic polynomial function using least square methods.

2. Given the position x of a point p, find the center of nearest cube that covers the point by computing $round(\frac{x_i}{L_i})$, $round(\frac{x_i}{L_i})$.

3. Compute the normalized relative position of p by computing Δx.
4. Use the quadratic polynomial to compute the approximated error.

We believe that the error is usually small. The cosine window algorithm is only used in *mspice* for smoothness.

Linear coefficient

Let us consider a simple case that *bbox* is covered by only one cube first. We want to have smaller error, therefore, it is an optimization problem:

$$\begin{aligned} min\ e &= |\bar{f}(x) - f(x)| \qquad (1.25)\\ f(x) &= u' \cdot A \cdot u \\ \bar{f}(x) &= b' \cdot u \end{aligned}$$

However, the L_1 norm optimization problem is difficult to solve, while the L_2 norm error problem is simple to handle. Therefore, we solve a simple problem instead

$$\begin{aligned} min\ E &= \int_{lo_1}^{hi_1} \cdots \int_{lo_n}^{hi_n} (\bar{f}(x) - f(x))^2\, dx_1 \cdots dx_n \qquad (1.26)\\ f(x) &= u' \cdot A \cdot u \\ \bar{f}(x) &= b' \cdot u \end{aligned}$$

The local minimal locates at where the derivative is zero.

$$\begin{aligned} \frac{dE}{db_i} &= 2\int_{lo_1}^{hi_1} \cdots \int_{lo_n}^{hi_n} (\bar{f}(x) - f(x)) x_i\, dx_1 \cdots dx_n \\ &= 0 \end{aligned}$$

Therefore,

$$\int_{lo_1}^{hi_1} \cdots \int_{lo_n}^{hi_n} \bar{f}(x) x_i\, dx_1 \cdots dx_n = \int_{lo_1}^{hi_1} \cdots \int_{lo_n}^{hi_n} f(x) x_i\, dx_1 \cdots dx_n$$

Now, let us rewrite it as a simple way

$$\begin{aligned} \oint_{Lo}^{Hi} \bar{f}(x) x_i\, dX &= \oint_{Lo}^{Hi} f(x) x_i\, dX \\ M(i,:) \cdot b &= c_i \end{aligned}$$

Finally, we have

$$\begin{aligned} M \cdot b &= c \\ b &= c \setminus M \qquad (1.27) \end{aligned}$$

If *bbox* is covered by multiple cubes, the computation of c vector is a little different, which is the sum of integrals of all cubes.

$$c_i = \sum_{cubes} \oint_{Lo}^{Hi} f(x) x_i\, dX$$

Computation of M matrix

First, let us consider the integral of polynomials.

$$\begin{aligned}
\oint_{Lo}^{Hi} x_1^{k_1} \cdots x_n^{k_n} \, dX &= \frac{\prod_{i=1}^{n} x_i^{(k_i+1)}|_{Lo}^{Hi}}{\prod_{i=1}^{n}(k_i+1)} \\
&= \prod_{i=1}^{n} \left(\frac{x_i^{k_i+1}}{k_i+1}|_{Lo_i}^{Hi_i}\right) \qquad (1.28) \\
&\equiv I(k_1, k_2, \cdots, k_n)
\end{aligned}$$

Obviously, we have the property

$$\oint_{Lo}^{Hi} (x_1^{k_1} \cdots x_n^{k_n}) \cdot (x_1^{m_1} \cdots x_n^{m_n}) \, dX = I(k_1+m_1, \cdots, k_n+m_n) \qquad (1.29)$$

Then, let us compute

$$
\begin{aligned}
M \cdot b &= \begin{bmatrix} \oint_{Lo}^{Hi} (b' \cdot u \cdot x_1)\, dX \\ \oint_{Lo}^{Hi} (b' \cdot u \cdot x_2)\, dX \\ \cdots\cdots\cdots\cdots \\ \oint_{Lo}^{Hi} (b' \cdot u \cdot x_n)\, dX \\ \oint_{Lo}^{Hi} (b' \cdot u)\, dX \end{bmatrix} \\
&\equiv \oint_{Lo}^{Hi} [(u \cdot * u') \cdot b]\, dX \quad \text{(commutative law)} \\
&= (\oint_{Lo}^{Hi} [(u \cdot * u')]\, dX) \cdot b \quad \text{(commutative law)} \\
M &= \oint_{Lo}^{Hi} [u \cdot * u']\, dX \quad \text{(matrix integral)} \\
&\equiv \begin{bmatrix} \oint_{Lo}^{Hi} x_1^2\, dX & \cdots & \oint_{Lo}^{Hi} x_1 x_n\, dX & \oint_{Lo}^{Hi} x_1\, dX \\ \cdots & \cdots & \cdots & \cdots \\ \oint_{Lo}^{Hi} x_1 x_n\, dX & \cdots & \oint_{Lo}^{Hi} x_n^2\, dX & \oint_{Lo}^{Hi} x_n\, dX \\ \oint_{Lo}^{Hi} x_1\, dX & \cdots & \oint_{Lo}^{Hi} x_n\, dX & \oint_{Lo}^{Hi} 1\, dX \end{bmatrix} \\
&= \begin{bmatrix} I(2,0,\cdots,0,0) & \cdots & I(1,0,\cdots,0,1) & I(1,0,\cdots,0,0) \\ \cdots & \cdots & \cdots & \cdots \\ I(1,0,\cdots,0,1) & \cdots & I(0,0,\cdots,0,2) & I(0,0,\cdots,0,1) \\ I(1,0,\cdots,0,0) & \cdots & I(0,0,\cdots,0,1) & I(0,0,\cdots,0,0) \end{bmatrix} \\
&\equiv \begin{bmatrix} I(e_1+e_1) & \cdots & I(e_n+e_1) & I(e_0+e_1) \\ \cdots & \cdots & \cdots & \cdots \\ I(e_1+e_n) & \cdots & I(e_n+e_n) & I(e_0+e_n) \\ I(e_1+e_0) & \cdots & I(e_n+e_0) & I(e_0+e_0) \end{bmatrix} \\
&\equiv I\left(\begin{bmatrix} (e_1+e_1) & \cdots & (e_n+e_1) & (e_0+e_1) \\ \cdots & \cdots & \cdots & \cdots \\ (e_1+e_n) & \cdots & (e_n+e_n) & (e_0+e_n) \\ (e_1+e_0) & \cdots & (e_n+e_0) & (e_0+e_0) \end{bmatrix}\right) \\
&\equiv I\left(\begin{bmatrix} e_1 & \cdots & e_1 & e_1 \\ \cdots & \cdots & \cdots & \cdots \\ e_n & \cdots & e_n & e_n \\ e_0 & \cdots & e_0 & e_0 \end{bmatrix} + \begin{bmatrix} e_1 & \cdots & e_n & e_0 \\ \cdots & \cdots & \cdots & \cdots \\ e_1 & \cdots & e_n & e_0 \\ e_1 & \cdots & e_n & e_0 \end{bmatrix}\right) \\
&\equiv I([repmat(E_n, 1, n+1) + repmat(\widehat{E_n}, n+1, 1)]) \quad (E = [e_1; \cdots; e_n; e_0] = [eye(n); 0]) \\
&\qquad (\widehat{E} = [e_1, \cdots, e_n, e_0] \\
&\equiv I(EE_n + \widehat{EE}_n) \qquad (1.30)
\end{aligned}
$$

Here, EE_n and $\widehat{EE}_n$ are two $(n+1) \times (n+1) \times n$ 3D array. The 3^{rd} dimension is for the n variables of I function. Of course, the 3^{rd} dimension can be treated as a cell. Then EE_n and $\widehat{EE}_n$ are $(n+1) \times (n+1)$ cell arrays, and each cell is a $1 \times n$ vectors as the parameters of I function. $I(EE_n)$ is defines as a

$(n+1)\times(n+1)$ array whose elements are $I(EE_n\{i,j\})$. It is easy to know that $I(EE_n) = I(\widehat{EE}_n)'$. Of course, $EE_n + \widehat{EE}_n$ is unfolded into $(n+1)*(n\cdot(n+1))$ array in the implementation.

Take 3D case as an example,

$$\begin{aligned}
M &= \begin{bmatrix} I(2,0,0) & I(1,1,0) & I(1,0,1) & I(1,0,0) \\ I(1,1,0) & I(0,2,0) & I(0,1,1) & I(0,1,0) \\ I(1,0,1) & I(0,1,1) & I(0,0,2) & I(0,0,1) \\ I(1,0,0) & I(0,1,0) & I(0,0,1) & I(0,0,0) \end{bmatrix} \\
&= I\left(\begin{bmatrix} (2,0,0) & (1,1,0) & (1,0,1) & (1,0,0) \\ (1,1,0) & (0,2,0) & (0,1,1) & (0,1,0) \\ (1,0,1) & (0,1,1) & (0,0,2) & (0,0,1) \\ (1,0,0) & (0,1,0) & (0,0,1) & (0,0,0) \end{bmatrix}\right) \\
&= I\left(\begin{bmatrix} (1,0,0) & (1,0,0) & (1,0,0) & (1,0,0) \\ (0,1,0) & (0,1,0) & (0,1,0) & (0,1,0) \\ (0,0,1) & (0,0,1) & (0,0,1) & (0,0,1) \\ (0,0,0) & (0,0,0) & (0,0,0) & (0,0,0) \end{bmatrix} + \begin{bmatrix} (1,0,0) & (0,1,0) & (0,0,1) & (0,0,0) \\ (1,0,0) & (0,1,0) & (0,0,1) & (0,0,0) \\ (1,0,0) & (0,1,0) & (0,0,1) & (0,0,0) \\ (1,0,0) & (0,1,0) & (0,0,1) & (0,0,0) \end{bmatrix}\right)
\end{aligned}$$

The pattern can be generated by matlab code

```
En   = [eye(dim);zeros(1,dim)];
EEn  = repmat(En,1,dim+1);
EEn2 = repmat(reshape(En',1,[]),dim+1,1);
pM   = EEn+EEn2; % add 1 for integral
```

Computation of c vector

Now, let us consider the computation of c vector. Similarly, let us consider the simple case with only one cube.

c is computed by integrating the quadratic polynomial in the cube. However, variables are changed with in the cube as shown in equation 1.19. Given $x = \varphi(y)$, we have

$$\begin{aligned}
\int_{x_l}^{x_h} f(x)\,dx &= \int_{\varphi^{-1}(x_l)}^{\varphi^{-1}(x_h)} f(\varphi(y))\,d\varphi(y) \\
&= \int_{y_l}^{y_h} f(\varphi(y))\,\varphi'(y)dy \\
\oint_{x_l}^{x_h} f(x)\,dx &= \oint_{y_l}^{y_h} f(\varphi(y))|\frac{\partial x}{\partial y}|\,dy
\end{aligned}$$

Therefore,

$$\oint_{Lo}^{Hi} f(x)dX = \prod_{i=1}^{n} L_i \cdot \int_{\Delta Lo}^{\Delta Hi} f(\Delta x)d\Delta X$$

In the following, we ignore this step and just add a $\prod_{i=1}^{n} L_i$ term to the final result.

First computing the constant term,

$$\begin{aligned}
c_{n+1} &= \oint_{Lo}^{Hi} f(x)\,dX \\
&= \oint_{Lo}^{Hi} (u' \cdot\!* B \cdot\!* u)\,dX \quad (u = [\Delta x; 1], \text{normalized relative position in quadratic function}) \\
&= \oint_{Lo}^{Hi} (sum(sum((u \cdot u') \cdot\!* B))\,dX \\
&= sum(sum([\oint_{Lo}^{Hi} (u \cdot u')\,dX] \cdot\!* B)) \quad (\text{commutative law}) \\
&= sum(sum([I(EE_n + \widehat{EE}_n)] \cdot\!* B)) \\
&= sum(sum(M \cdot\!* B))
\end{aligned}$$

It is noticed that it has similar pattern with the computation of M matrix[4]. Then computing other terms

$$\begin{aligned}
c_i &= \oint_{Lo}^{Hi} f(x) \cdot \hat{x}_i\,dX \quad (\text{use } \hat{x} \text{ to indicate the original position}) \\
&= \oint_{Lo}^{Hi} f(x) \cdot (x0_i + x_i) \cdot L_i\,dX \quad (\ x0 \text{ is the position of cube center }) \\
&= L_i \cdot \oint_{Lo}^{Hi} f(x) \cdot (x0_i + x_i)\,dX \\
&= L_i \cdot (x0_i \cdot c_{n+1} + \oint_{Lo}^{Hi} f(x) \cdot x_i\,dX)
\end{aligned}$$

and

$$\begin{aligned}
\oint_{Lo}^{Hi} f(x) \cdot x_i\,dX &= sum(sum([\oint_{Lo}^{Hi} (u \cdot u' \cdot x_i)\,dX] \cdot\!* B)) \\
&= sum(sum(I(EE_n + \widehat{EE}_n + e_i) \cdot\!* B)) \quad (\text{use equation 1.29})
\end{aligned}$$

Therefore,

$$c = L' \cdot \left[\begin{bmatrix} sum(sum(\ I(EE_n + \widehat{EE}_n + e_1) \cdot\!* B\ )) \\ \cdots\cdots\cdots\cdots\cdots\cdots \\ sum(sum(\ I(EE_n + \widehat{EE}_n + e_n) \cdot\!* B\ )) \\ sum(sum(\ I(EE_n + \widehat{EE}_n + e_0) \cdot\!* B\ )) \end{bmatrix} + c_{n+1} \cdot x0 \right]$$

[4] B is an upper triangular matrix, we only store the non zero elements. Therefore, we should trim $u \cdot u'$ in the implementation.

As we mentioned, we use B instead of A to save space. And B is an upper triangular matrix. The upper triangular elements of B is save as a vector. Therefore, it is not necessary to compute the all integral of $EE_n + \widehat{EE}_n + e_i$. Only the lower triangular elements are computed [5]. Therefore

$$\begin{aligned}
c &= L' \cdot \left[\begin{bmatrix} sum(\ I(tril(EE_n + \widehat{EE}_n) + e_1) \cdot\!* \vec{B}\ ) \\ \cdots\cdots\cdots\cdots\cdots\cdots \\ sum(\ I(tril(EE_n + \widehat{EE}_n) + e_n) \cdot\!* \vec{B}\ ) \\ sum(\ I(tril(EE_n + \widehat{EE}_n) + e_0) \cdot\!* \vec{B}\ ) \end{bmatrix} + c_{n+1} \cdot x0 \right] \\
&= L' \cdot \left[\begin{bmatrix} I(tril(EE_n + \widehat{EE}_n) + e_1)' \\ \cdots\cdots\cdots\cdots \\ I(tril(EE_n + \widehat{EE}_n) + e_n)' \\ I(tril(EE_n + \widehat{EE}_n) + e_0)' \end{bmatrix} \cdot \vec{B} + c_{n+1} \cdot x0 \right] \\
&= L' \cdot \left[I \left(\begin{bmatrix} tril(EE_n + \widehat{EE}_n)' + e_1 \\ \cdots\cdots\cdots\cdots \\ tril(EE_n + \widehat{EE}_n)' + e_n \\ tril(EE_n + \widehat{EE}_n)' + e_0 \end{bmatrix} \right) \cdot \vec{B} + c_{n+1} \cdot x0 \right] \qquad (1.31)
\end{aligned}$$

Finally, c is the sum of all cubes

$$c = \prod_{i=1}^{n} L_i \cdot \sum_{cube=1}^{nc} c_{cube}$$

The pattern can be generated by the matlab code

```
cpM  = mat2cell(pM,ones(1,n+1),n*ones(1,n+1));
pc0 = cpM(qpind);
pc0 = cell2mat(pc0');
pc = [repmat(pc0,n,1)+repmat(eye(n),1,k); pc0];
```

Computation of error bound *err*

Now, let us compute the maximum error over all cubes. It is obviously that the maximum error is the maximum of errors for each cube. Therefore, let us consider only one cube. The error is defined as

$$e \quad = \quad \bar{f}(x) - f(x)$$

The extreme points locates at critical points where derivative is zero or

[5] In matlab, matrix is column first. But I prefer row first order of matrix. The order of lower triangle is the same with the order used to save B.

boundary of the cube. Let us compute the derivative first.

$$\begin{aligned}
\frac{de}{dx_i} &= \frac{d\bar{f}(x)}{dx_i} - \frac{df(x)}{d(x_i)} \\
b_i &= \frac{df(\Delta x_i)}{d\Delta x_i} \cdot \frac{d\Delta x_i}{dx_i} \\
&= \frac{1}{L_i} \cdot \frac{df(\Delta x)}{d\Delta x_i} \quad \text{(variable in cube, ignore } \Delta \text{ below)} \\
b_i \cdot L_i &= sum\left(sum\left(\frac{d[u \cdot u']}{dx_i} \cdot * B\right)\right) \\
&= sum\left(sum\left(d\begin{bmatrix} x_1^2 & \cdots & x_1x_n & x_1 \\ \cdots & \cdots & \cdots & \cdots \\ x_nx_1 & \cdots & x_n^2 & x_n \\ x_1 & \cdots & x_n & 1 \end{bmatrix} / dx_i \cdot * B\right)\right) \\
&= sum\left(sum\left(\begin{bmatrix} 0 & \cdots & x_1 & \cdots & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ x_1 & \cdots & 2x_i & \cdots & x_n & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & x_n & \cdots & 0 & 0 \\ 0 & \cdots & 1 & \cdots & 0 & 0 \end{bmatrix} \cdot * B\right)\right) \\
&= \begin{bmatrix} x_1 & \cdots & 2x_i & \cdots & x_n & 1 \end{bmatrix} \cdot \widehat{B}(:, i) \quad (\widehat{B} \text{ is full form of } B) \\
&= \left[\begin{bmatrix} 1 & \cdots & 2 & \cdots & 1 & 1 \end{bmatrix} \cdot * \widehat{B}(i, :)\right] \cdot u
\end{aligned}$$

Totally, we have

$$\begin{aligned}
\left[\begin{bmatrix} 2 & \cdots & 1 & 1 \\ \cdots & \cdots & \cdots & \cdots \\ 1 & \cdots & 2 & 1 \\ 1 & \cdots & 1 & 2 \end{bmatrix} \cdot * \widehat{B}\right] \cdot u &= L \cdot * b \\
\left[[1 + eye(n+1)] \cdot * \widehat{B}\right] \cdot u &= L \cdot * b
\end{aligned}$$

Of course, the last row is nonsense which is not use to solve the linear system.

$$[(1 + eye(n)) \cdot * \widehat{B}(1:n, 1:n)] \cdot x = L(1:n) \cdot * b(1:n) - \widehat{B}(1:n, n+1) \tag{1.32}$$

The result is easier to explain using A, as we know

$$\frac{d(x' * A * x)}{dx} = 2Ax \tag{1.33}$$

Take 3D for example, the equation is

$$\begin{bmatrix} 2b_1 & b_2 & b_3 & b_4 \\ b_2 & 2b_5 & b_6 & b_7 \\ b_3 & b_6 & 2b_8 & b_9 \\ b_4 & b_7 & b_9 & 2b_{10} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{bmatrix} = \begin{bmatrix} L_1 b_1 \\ L_2 b_2 \\ L_3 b_3 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 2b_1 & b_2 & b_3 \\ b_2 & 2b_5 & b_6 \\ b_3 & b_6 & 2b_8 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} L_1 b_1 \\ L_2 b_2 \\ L_3 b_3 \end{bmatrix} - \begin{bmatrix} b_4 \\ b_7 \\ b_9 \end{bmatrix}$$

The we should find the boundary, which referrs to the $2n$ $n-1$ dimensional faces. Similarly, the extreme points on the face may locate at critical points or boundary. Let us compute the derivative for the face, for example $x_1 = Lo_1$ face. The computation of derivative is similar with above, except that the variable x_1 is fixed to the value of Lo_1. Therefore, $\frac{de}{dx_1}$ should be removed from the linear systems. And x_1 should be replace with Lo_i also, which can be done by add the constraint $x_1 = Lo_i$ to the linear system. Therefore, the linear system is the same with equation 1.32 except replacing the i^{th} constraint by its fixed value.

$$\left[\begin{bmatrix} 1 & \cdots & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots \\ 1 & \cdots & 2 & 1 \\ 1 & \cdots & 1 & 2 \end{bmatrix} \cdot * \widehat{B} \right] \cdot u = \begin{bmatrix} \widehat{B}(1,1) lo_1 \\ \vdots \\ L_n b_n \\ 1 \end{bmatrix}$$

Generally, the critical points of a $n-k$ face can be computed by solving a linear system which replaces k corresponding constraint of equation 1.32 by its fixed value. Especially, the critical points of a 0 dimensional face (i.e, vertices of cube) is the linear system that replaces all constraint by fixed value.

$$\begin{bmatrix} 1 & \cdots & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & 1 & 0 \\ 1 & \cdots & 1 & 2 \end{bmatrix} \cdot u = \begin{bmatrix} lo_1 \\ \vdots \\ lo_n \\ 1 \end{bmatrix}$$

Therefore, for each constraint, there are three choice 1) constraint of critical point 2) constraint of low boundary 3) constraint of high boundary.

$$\left.\begin{matrix} \left[\begin{bmatrix} 1 & \cdots & 2 & \cdots & 1 \end{bmatrix} \cdot * \widehat{B}(i, 1:n) \right] \\ \begin{bmatrix} 0 & \cdots & 1 & \cdots & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & \cdots & 1 & \cdots & 0 \end{bmatrix} \end{matrix} \right\} \cdot x = \left\{ \begin{matrix} L_i b_i - \widehat{B}(i, n+1) \\ lo_i \\ hi_i \end{matrix} \right. \tag{1.34}$$

And if all of 3^n linear systems are solved, all critical points and boundary points (vertices of cube) are found. Thus the maximum and minimum error can be computed by checking these points. Of course, the boundary points can be accessed directly without solving the linear system, which can be used to optimized the performance.

The implementation

The final code is

```
function [b,err] = linearizeQuadFit(model,bbox,useDirectVertices)
if(any(bbox(:,2)<bbox(:,1)))
    error('infeasible region');
end;
if(nargin<3||isempty(useDirectVertices))
    useDirectVertices = true;
end;

% special case when l = h
ind = find(bbox(:,2)==bbox(:,1));
bbox(ind,:) = bbox(ind,:)+repmat([-1e-6,1e-6],length(ind),1);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% compute dimension or model related data
n = length(model.nv);
k = (n+1)*(n+2)/2; % # of quadratic parameters
qpind = tril(true(n+1,n+1));

grid = bbox ./ [model.dv,model.dv] - round([model.v0,model.v0]./[model.dv,model.dv]);
bboxm = [round(grid(:,1)), round(grid(:,2))-(mod(grid(:,2),1)==0.5)];
nz = (bboxm(:,2)-bboxm(:,1))+1;
nc = prod(nz); % # of cubes

L = zeros(n,nc);
H = zeros(n,nc);
S = zeros(n,nc);
ind = cell(n,1);
for i=1:n
    SIZ = reshape(nz,1,[]); SIZ(i) = 1;
    SIZ2 = ones(size(SIZ)); SIZ2(i) = nz(i);
    L(i,:) = reshape(repmat(reshape([grid(i,1)-bboxm(i,1),repmat(-0.5,1,nz(i)-1)],SIZ2),SIZ),1,[]);
    H(i,:) = reshape(repmat(reshape([repmat(0.5,1,nz(i)-1),grid(i,end)-bboxm(i,2)],SIZ2),SIZ),1,[]);
    S(i,:) = reshape(repmat(reshape(model.shift{i}(bboxm(i,1):bboxm(i,2)),SIZ2),SIZ),1,[]);
    ind(i) = {bboxm(i,1):bboxm(i,2)};
end;
MP = reshape(model.data(ind{:}),[],1);
MP = reshape(cell2mat(MP),[],nc);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Compute M matrix
En   = [eye(n);zeros(1,n)];
EEn  = repmat(En,1,n+1);
EEn2 = repmat(reshape(En',1,[]),n+1,1);
pM   = EEn+EEn2;
cpM  = mat2cell(pM,ones(1,n+1),n*ones(1,n+1));

diffs = [diff(bbox,[],2), diff(bbox.^2,[],2)/2, diff(bbox.^3,[],2)/3];
dind = sub2ind([n,3],repmat(1:n,n+1,n+1),pM+1);
M = reshape(prod(reshape(diffs(dind)',n,[]),1),n+1,n+1)';

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Compute the c vector
pc0 = cpM(qpind);
pc0 = cell2mat(pc0');
pc = [repmat(pc0,n,1)+repmat(eye(n),1,k); pc0];

diffs = [H-L; (H.^2-L.^2)/2; (H.^3-L.^3)/3; (H.^4-L.^4)/4];
dind = sub2ind([n,4],repmat(1:n,n+1,k),pc+1);
dind = reshape(dind',[],1);
dind = sub2ind([4*n,nc],repmat(dind,1,nc),repmat(1:nc,(n+1)*k*n,1));

I = diffs(dind);
I = prod(reshape(I,n,[]),1);
I = reshape(I,[],nc);
```

```

mMP = repmat(MP,n+1,1);
c = mMP.*I;
c = reshape(sum(reshape(c,k,[]),1),[],nc);
c(1:n,:) = c(1:n,:) + repmat(c(end,:),n,1).*S;
c = sum(c,2);

c = c.*([model.dv;1]);
c = c.*prod(model.dv);

% Finally, find the best fit
b = M\c;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Compute the error
choice = 3; np = choice^n;

% compute the pattern
pos = zeros(n+1,n+1);
pos(qpind) = 1:k;
pos = pos+pos'-diag(diag(pos));
posl = pos(1:n,1:n); posr = pos(1:n,end);

% prepare the linear system: posl.*(ones+eye)*x = dv*b - posr
C = repmat(model.dv.*b(1:n),1,nc)  - MP(posr,:);
A = MP(posl,:).*repmat(reshape(1+eye(n),[],1),1,nc);
A = reshape(A,n,[]);

C = [C,L,H];
A = [A,repmat(eye(n),1,2*nc)];
cC = mat2cell(C,repmat(1,1,n),repmat(nc,1,choice));
cA = mat2cell(A,repmat(1,1,n),repmat(n*nc,1,choice));

% compute index of all combinations
pInd = mod(floor(   repmat((0:(np-1)),n,1)...
                 ./repmat(choice.^(((n-1):-1:0)'),1,np) ), choice );
if(useDirectVertices)
    nvInd = (prod(pInd,1)==0); % non-vertex index
    vInd = ~nvInd; % vertex index
    nv = (choice-1)^n; nnv = np - nv;
end;

pInd = sub2ind([n,choice],repmat((1:n)',1,np),pInd+1);
CC = cell2mat(cC(pInd));
AA = cell2mat(cA(pInd));

% compute all critical points
if(useDirectVertices)
    nvAA = AA(:,reshape(repmat(nvInd,n*nc,1),1,[]));
    nvCC = CC(:,reshape(repmat(nvInd,nc,1),1,[]));
    nvx  = solveLinearSystems(nvAA,nvCC);
    vx   = CC(:,reshape(repmat(vInd,nc,1),1,[]));
    x = [nvx,vx];
else
    x = solveLinearSystems(AA,CC);
end;

% evaluate the error over all points
u = [x;ones(1,nc*np)];
dv = reshape(repmat(u,n+1,1),n+1,[]).*repmat(reshape(u,[],1),1,n+1)';
dv = reshape(dv,(n+1)^2,[]);
dv = dv(qpind,:);
realV = repmat(MP,1,np).*dv;
realV = sum(realV,1);
xx = (x+repmat(S,1,np)).*repmat(model.dv,1,nc*np);
estV = b(1:end-1)'*xx+b(end);
errFull = estV - realV;

```

```
133 % However, some points are not in the cube, remove it.
134 if(useDirectVertices)
135     inCube = [( all(nvx >= repmat(L,1,nnv),1) & all(nvx <= repmat(H,1,nnv),1) ), true(1,nv*nc)];
136 else
137     inCube = [( all(x >= repmat(L,1,np),1) & all(x <= repmat(H,1,np),1) )];
138 end;
139 errFull = errFull(inCube);
140
141 err = [min(errFull); max(errFull)];
142
143 % Make the error balance
144 b(end) = b(end)-(mean(err));
145 err = (err(2)-err(1))/2;
```

When $Lo_i = Hi_i$, the integral is zero, thus $M = 0$ and $c = 0$. We increase the *bbox* by a small amount to handle the case on line $10 - 11$.

Line $15 - 17$ compute the number of variables (dimensions) n, number of coefficient of quadratic polynomial k and their indices in B matrix [6].

Line $19 - 22$ maps *bbox* into to cubes. *round* function is applied to $model.v0$ because of round-off error. We use $round(0.5) = 0$ for upper bound to avoid the $n - 1$ dimensional cubes. nc is the number of cubes.

The for loop on lines $28 - 35$ compute the lower bound and upper bound of each cube, and also their center position. However S is not $x0$ in equation 1.19, it is $\frac{x0}{L}$. The structure of L, H, S is shown in figure 1.5. The indices of cubes is

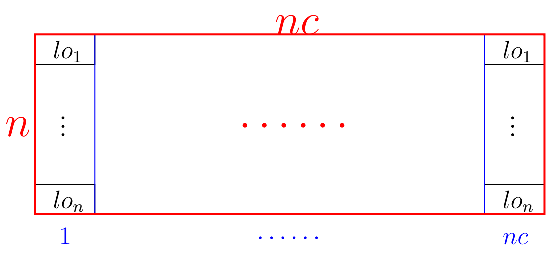


Figure 1.5: The structure of L, H, S matrix

also computed in the loop and the quadratic parameters are also extracted as MP, the structure is shown in figure 1.6

Lines $41 - 45$ compute the pattern pM for $\oint_{Lo}^{hi}[u \cdot * u']dX$ as shown in equation 1.30. cpM is the cell version. The structure is shown in figure 1.7.

Line 47 computes $\frac{x_i^{k_i+1}}{k_i+1}$ as shown in equation 1.29. It has the structure as

$$\begin{bmatrix} \Delta x_1 & \Delta x_1^2/2 & \Delta x_1^3/3 \\ \multicolumn{3}{c}{\dotfill} \\ \Delta x_n & \Delta x_n^2/2 & \Delta x_n^3/3 \end{bmatrix}$$

Line 48 computes the indices of $diffs$ for each integral of pM, 1 is added to PM for integral. And line 49 pickup the elements, compute the product for

[6]Column first in matlab, thus us lower triangular matrix

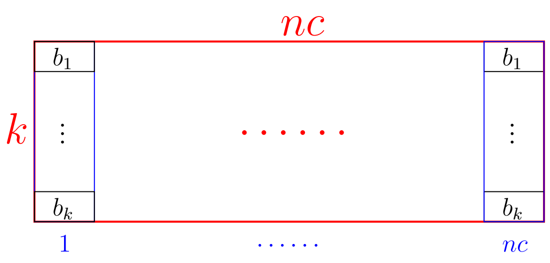


Figure 1.6: The structure of MP matrix

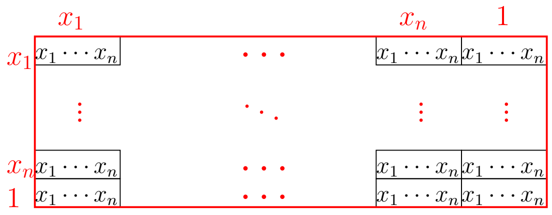


Figure 1.7: The structure of pM matrix

each cell as shown in figure 1.8. We use transpose twice to move each cell (row first order) to a column because of column first order in matlab.

Lines $53 - 55$ compute the pattern of integral used in c vector in a cube as shown in equation 1.31. Line 53 picks up the lower triangular cells for $tril(EE_n + \widehat{EE}_n)'$, each column of $pc0$ is for one integral. Line 55 compute the pattern for c and add e_i for c_i. The structure is shown in figure 1.9.

Line 57 compute the integral term for each cube as

$$\begin{bmatrix} \begin{bmatrix} (\Delta x_1^1)^1/1 & \cdots & (\Delta x_n^1)^1/1 \\ \cdots & \ddots & \cdots \\ (\Delta x_1^{nc})^1/1 & \cdots & (\Delta_n^{nc})^1/1 \end{bmatrix} \\ \begin{bmatrix} (\Delta x_1^1)^2/2 & \cdots & (\Delta x_n^1)^2/2 \\ \cdots & \ddots & \cdots \\ (\Delta x_1^{nc})^2/2 & \cdots & (\Delta_n^{nc})^2/2 \end{bmatrix} \\ \begin{bmatrix} (\Delta x_1^1)^3/3 & \cdots & (\Delta x_n^1)^3/3 \\ \cdots & \ddots & \cdots \\ (\Delta x_1^{nc})^3/3 & \cdots & (\Delta_n^{nc})^3/3 \end{bmatrix} \\ \begin{bmatrix} (\Delta x_1^1)^4/4 & \cdots & (\Delta x_n^1)^4/4 \\ \cdots & \ddots & \cdots \\ (\Delta x_1^{nc})^4/4 & \cdots & (\Delta_n^{nc})^4/4 \end{bmatrix} \end{bmatrix}$$

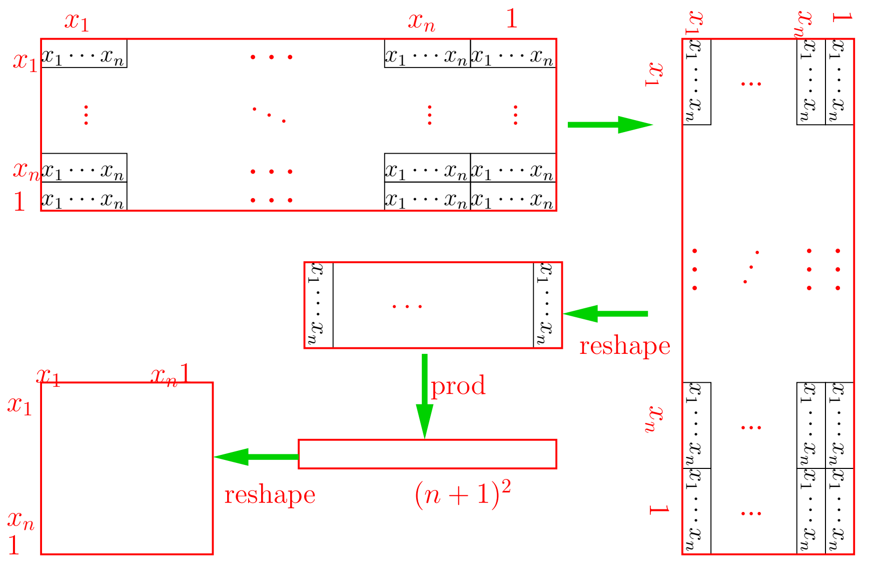


Figure 1.8: The computation of M matrix

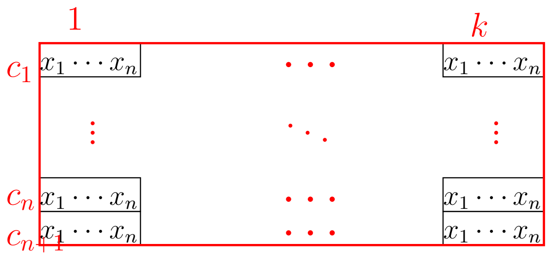


Figure 1.9: The structure of pc matrix

Line 58 computes the column position (within a cube) of $diffs$ for each integral of pc. Line 59 uses transpose to place all integral for c_i together in a column. Line 60 computes the indices of $diffs$ for all cubes.

Line $62 - 64$ compute the integral for the c vector. The process is shown in figure 1.10.

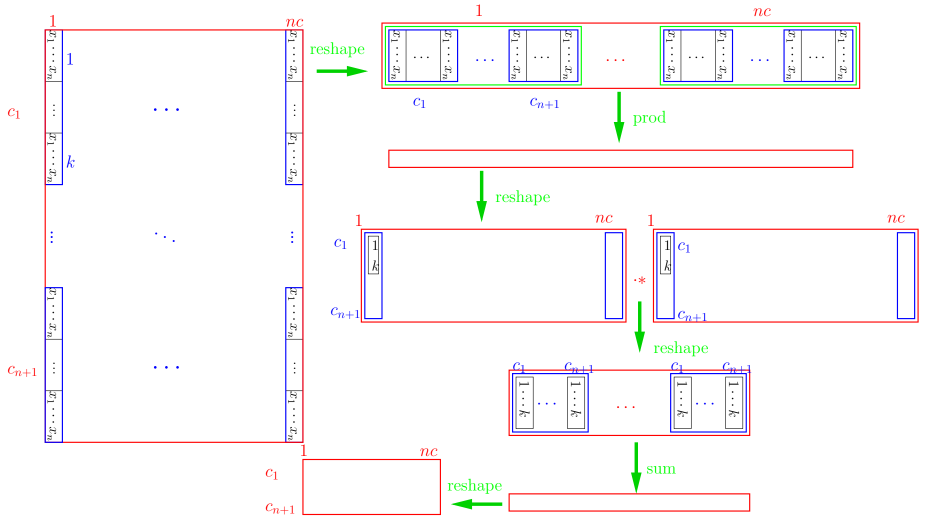


Figure 1.10: The computation of c vector

Lines $66 - 67$ get the quadratic polynominal parameters and compute the polynomial, with each column for one cube. Line 68 computes the sum for c_i and line 69 adds c_{n+1} to c_i using S. Line 70 computes the sum over all cubes. Line 72 times c_i by L_i and line 73 times the final result by $\prod_{i=1}^{n} L_i$.

Finally, line 76 computes the linear coefficient b.

Lines $83 - 86$ computes indices of $\widehat{B}$. Lines $89 - 91$ constructs the linear system as shown in equation 1.32. C is a $n \times nc$ matrix, each column for a cube. A is a $n \times (n * nc)$ matrix, each $n \times n$ for one linear system.

Lines $93-96$ all three choices as shown in equation 1.34. cC and cA converts A and C to $n \times 3$ cell array, each cell for one variable x_i of all cubes.

Line 99 compute the indices of all 3^n combinations. Take 3D for example, the result $pInd$ is

```
0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2
0 0 0 1 1 1 2 2 2 0 0 0 1 1 1 2 2 2 0 0 0 1 1 1 2 2 2
0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2
```

where each column is a combination. For example, the sixth column mean we use first choice for x_1, low value for x_2 and high value for x_3. Line 107 computes the indices of all choices. Lines $108 - 109$ generates linear systems of all combinations.The structure of CC is as

$$\begin{bmatrix} \begin{bmatrix} c_1^1 & \cdots & c_1^{nc} \\ \vdots & \ddots & \vdots \\ c_n^1 & \cdots & c_n^{nc} \end{bmatrix} & \cdots & \begin{bmatrix} c_1^1 & \cdots & c_1^{nc} \\ \vdots & \ddots & \vdots \\ c_n^1 & \cdots & c_n^{nc} \end{bmatrix} \\ 1 & \cdots & np \end{bmatrix}$$

Line 119 solves all linear systems and finds all critical points.

Lines 123−128 compute the quadratic function value on these critical points, line 124 computes $u \cdot u'$, line 126 picks up the lower triangular elements for all cubes, lines 127 − 128 computes $tril(u \cdot u') \cdot \vec{B}$. Lines 129 − 130 compute the linear approximation value on these critical points.

However, not all critical points are within the cube, therefore, line 137 finds the indices of critical points which are in the cube.

And finally the maximum and minium values are found on line 141 and the error are balanced at the end.

To speedup computation, we notices that the 2^n cube vertices can be used directly without solving a linear system. Therefore, lines 101 − 105 finds indices of cube vertices. Lines 113 − 117 picks up the necessary linear systems and solves them, the cube vertices is also appended. And line 135 only find indices of non-vertex critical points which are in the cubes, all vertices are set to true because it must be in the cube.

1.2.3 Lipschitz Method

It is found that most of time (about 90%)is spent on error computation. Solving linear systems cost about two third of total time. Therefore, it is necessary to speedup the computation of error.

For each cube, the error is also a quadratic polynomial

$$\begin{aligned} e &= f(x) - \hat{f}(x) \\ &= u' \cdot A \cdot u - b \cdot \hat{u} \quad (\hat{x_i} = (x_i + x0_i) \cdot L_i) \\ &= u' \cdot eA \cdot u \end{aligned}$$

Lipschitz continuous

Obviously, the function is *Lipschitz continuous* with the property

$$f(x1) - f(x2) \le K \cdot |x1 - x2|$$

where K is the *Lipschitz constant*. Therefore, we can evaluate the function on the center point and approximate the error bound using the Lipschitz constant.

For the quadratic polynomial, it is known that the derivative [7] is

$$\frac{d(u' \cdot A \cdot u)}{du} = 2 \cdot A \cdot u$$

[7] see equation 1.32

Of course, $\frac{de}{dx_{n+1}} = 0$.

For $u \in u0 \pm r$, $A \cdot u$ is bounded by

$$\begin{aligned} A \cdot u &= A \cdot \begin{bmatrix} u0_1 \pm r_1 \\ \cdots \\ u_{n+1} \pm r_{r+1} \end{bmatrix} \\ &\in [A \cdot u0 - abs(A) \cdot r, A \cdot u0 + abs(A) \cdot r] \end{aligned}$$

Totally, we have

$$e(u) \in e(u0) \pm 2(A \cdot u + abs(A) \cdot r)' \cdot r \tag{1.35}$$

The code in implemented as

```
 1 function [ymin, ymax] = quadBounds(As,x0s,rs,y0s)
 2 [n,np] = size(x0s);
 3 u0s = [ x0s; ones(1,np)];
 4
 5 if(nargin<4||isempty(y0s))
 6     dv = repmat(u0s,n+1,1).*reshape(repmat(reshape(u0s,[],1),1,n+1)',(n+1)^2,[]);
 7     y0s = sum(As.*dv,1);
 8 end;
 9
10 % compute derivative
11 dy0s = 2*reshape(sum(reshape(As.*repmat(u0s,n+1,1),n+1,[]),1),n+1,[]); % 2*A*u
12 dy0s = dy0s(1:n,:);
13
14 rAs = reshape(As,n+1,[]);
15 hs = reshape(sum(abs(rAs(1:n,:)),1),n+1,[]);
16 hs = 2*hs(1:n,:);
17 hs = hs.*rs;
18
19 dysmin = dy0s - hs;
20 dysmax = dy0s + hs;
21
22 % find the bound
23 dy = sum(rs.*max(abs(dysmin),abs(dysmax)),1);
24 ymin = y0s - dy;
25 ymax = y0s + dy;
```

The function is vectorized for large number of cubes. Each row of $As, x0s, rs, y0s$ for is for a cube. Lines $6-7$ computes $f(x0)$ if not provided. Lines $11-12$ evaluate the derivative on center point $\frac{df(x0)}{dx}$. The constant term is removed from $2 \cdot A \cdot u$. Lines $14-17$ over approximates the change of derivative (the constant term is removed). Therefore, the maximum and minimum derivative over the cube is computed on lines $19-20$ and thus the range of $f(x)$ is avaliable at the end.

Iteration method

However, the upper bound might be much larger than the real value. Our solution is to divide a cube into (2^n) smaller ones to reduce the approximation error. Therefore, an lower bound is required to know the distance between approximation and real value. A good lower bound is computed by evaluating the function on grid points and the center point of each cube. Therefore, the cube is splitted into smaller one until the gap between upper bound and lower bound is small.

The final code is

```
function [b,err] = linearizeQuadFitLip(model,bbox,tol)
if(any(bbox(:,2)<bbox(:,1)))
    error('infeasible region');
end;
if(nargin<3||isempty(tol))
    tol = 0.02; % 2 percent
end;

ind = find(bbox(:,2)==bbox(:,1));
bbox(ind,:) = bbox(ind,:)+repmat([-1e-6,1e-6],length(ind),1);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% compute dimension or model related data
n = length(model.nv);
k = (n+1)*(n+2)/2; % # of quadratic parameters
qpind = tril(true(n+1,n+1)); % lower triangular elements, the order save in model.data.

grid = bbox ./ [model.dv,model.dv] - round([model.v0,model.v0]./[model.dv,model.dv]); %round off
bboxm = [round(grid(:,1)), round(grid(:,2))-(mod(grid(:,2),1)==0.5)]; % avoid empty cube because round(0.5)=1
nz = (bboxm(:,2)-bboxm(:,1))+1;
nc = prod(nz); % # of cubes

L = zeros(n,nc);
H = zeros(n,nc);
S = zeros(n,nc); % center of cube
ind = cell(n,1);
for i=1:n
    SIZ = reshape(nz,1,[]); SIZ(i) = 1;
    SIZ2 = ones(size(SIZ)); SIZ2(i) = nz(i);
    L(i,:) = reshape(repmat(reshape([grid(i,1)-bboxm(i,1),repmat(-0.5,1,nz(i)-1)],SIZ2),SIZ),1,[]);
    H(i,:) = reshape(repmat(reshape([repmat(0.5,1,nz(i)-1),grid(i,end)-bboxm(i,2)],SIZ2),SIZ),1,[]);
    S(i,:) = reshape(repmat(reshape(model.shift{i}(bboxm(i,1):bboxm(i,2)),SIZ2),SIZ),1,[]);
    ind(i) = {bboxm(i,1):bboxm(i,2)};
end;
Bs = reshape(model.data(ind{:}),[],1);
Bs = reshape(cell2mat(Bs),[],nc); % (kxnc) one column for a cube.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Compute the best linear approximation to minize L2 norm error
 % Compute M matrix
En   = [eye(n);zeros(1,n)];
EEn  = repmat(En,1,n+1);
EEn2 = repmat(reshape(En',1,[]),n+1,1);
pM   = EEn+EEn2; % add 1 for integral
cpM  = mat2cell(pM,ones(1,n+1),n*ones(1,n+1)); % (n+1)x(n+1) cell, each cell is 1xn vector.

diffs = [diff(bbox,[],2), diff(bbox.^2,[],2)/2, diff(bbox.^3,[],2)/3]; %[dX, dX^2/2, dX^3/3]
dind = sub2ind([n,3],repmat(1:n,n+1,n+1),pM+1); % compute the indices of diffs used in M
M = reshape(prod(reshape(diffs(dind)',n,[]),1),n+1,n+1)';% prod from x_1 to x_n

% Compute the c vector
pc0 = cpM(qpind); % pickup the lower triangle elements. kx1 cell;
pc0 = cell2mat(pc0'); % pattern for c_{n+1}
pc = [repmat(pc0,n,1)+repmat(eye(n),1,k); pc0]; % pattern for c, add e_i.

diffs = [H-L; (H.^2-L.^2)/2; (H.^3-L.^3)/3; (H.^4-L.^4)/4]; % [dX; dX^2/4; dX^3/3; dX^4/4]
dind = sub2ind([n,4],repmat(1:n,n+1,k),pc+1); % indices of within each column (cube)
dind = reshape(dind',[],1); % tranpose to place each row of $c$ together
dind = sub2ind([4*n,nc],repmat(dind,1,nc),repmat(1:nc,(n+1)*k*n,1));% indices of all cubes, column for cube

I = diffs(dind); % pickup all integral terms
I = prod(reshape(I,n,[]),1); % compute the prod from x_1 to x_n
I = reshape(I,[],nc); % column for one cube

```

```
c = repmat(Bs,n+1,1).*I; % times by parameters
c = reshape(sum(reshape(c,k,[]),1),[],nc); % sum over all k polynomial terms
c(1:n,:) = c(1:n,:) + repmat(c(end,:),n,1).*S;  % add c_{n+1} to c_i
c = sum(c,2); % sum over all cubes

c = c.*([model.dv;1]); % cube length
c = c.*prod(model.dv); % constant term because of changing variable

% Finally, find the best fit
b = M\c;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Compute the error
% f(x) = u'*B*u
% ff(x) = b'[xx;1]; xx = (x+s).*model.dv
% e = f-ff = u'*eA*u;

% map indices from B to full version \hat{B};
map = zeros(n+1,n+1);
map(qpind) = 1:k;  map = map+map'-diag(diag(map));
cind = map(:,end);

% symmetric matrix eA
eBs = Bs;
xdv = repmat(b(1:end-1).*model.dv,1,nc); % b_i*dv_i
eBs(cind(1:end-1),:) = eBs(cind(1:end-1),:) - xdv; % -b_i*x_i*dv_i
eBs(cind(end),:) = eBs(cind(end),:) - (b(end) + sum(xdv.*S,1)); % constant term
eAs = repmat(reshape((1+eye(n+1))/2,[],1),1,nc).*eBs(map,:); % make it symmetric

% evaluate error for all grid points
% Notice: e is not continous over cubes because cosine window is ignored.
choice = 2; np = choice^n;
cV = mat2cell([L,H],ones(1,n),nc*ones(1,choice));
% indices of bbox[lo_1,hi_1;...lo_n,hi_n] for np vertices of each cube.
vind = mod(floor( repmat(0:(np-1),n,1)./repmat(choice.^(((n-1):-1:0)'),1,np) ), choice);
vind = sub2ind([n,choice],repmat((1:n)',1,np),vind+1);
x = cell2mat(cV(vind));

% u'*eA*u
u = [x; ones(1,nc*np)];
dv = repmat(u,n+1,1).*reshape(repmat(reshape(u,[],1),1,n+1)',(n+1)^2,[]);
lerr = sum(repmat(eAs,1,np).*dv,1);
lerr = [min(lerr);max(lerr)]; % max interval of all points

% split cubes into smaller one until gap between lower and upper bound is small
while(true)
    x0 = (L+H)/2; ds = (H-L)/2;

    % compute lower bound
    u = [x0;ones(1,nc)];
    dv = repmat(u,n+1,1).*reshape(repmat(reshape(u,[],1),1,n+1)',(n+1)^2,[]);
    e0 = sum(eAs.*dv,1);
    lerr = [ min(lerr(1),min(e0,[],2)); max(lerr(2),max(e0,[],2)) ];

    % compute upper bound
    [uemin,uemax] = quadBounds(eAs,x0,ds,e0);
    uerr = [min(uemin);max(uemax)];

    r   = diff(uerr)/diff(lerr);
    if(r>1+tol)
        % remove cube with smaller error than lower bound
        ind = (uemin <lerr(1) | uemax >lerr(2));
        L = L(:,ind); %H = H(:,ind);
        x0 = x0(:,ind); ds = ds(:,ind);
        eAs = eAs(:,ind);
        nc = size(L,2);

        % split each cube to 2^n small ones
```

```
133         newL = mat2cell([L,x0],ones(1,n),nc*ones(1,choice));
134         L = cell2mat(newL(vind));
135         H = L+repmat(ds,1,np);
136         eAs = repmat(eAs,1,np);
137         nc = nc*np;
138     else
139         err = uerr; % over approximated
140         break;
141     end;
142 end;
143
144 b(end) = b(end)+(mean(err));
145 err = diff(err)/2;
```

Lines $1-75$ is the same with the previous code, except 1) rename MP to Bs 2) add a new parameter *tol* as the maximum approximation error.

Lines $82-92$ compute the symmetric matrix for the error function. *map* is the indices of $\vec{B}$ for $\widehat{B}$ matrix. Therefore,

$$\begin{aligned} \widehat{B} &= \vec{B}(map) \\ A &= (1+eye(n+1))/2 \cdot * \widehat{B} \end{aligned}$$

For the error function, we have

$$\begin{aligned}
e &= sum(sum(B \cdot * u \cdot u')) - b' \cdot u \\
&= sum\left(sum\left(\begin{bmatrix} \vec{B}_1 x_1^2 & 0 & \cdots & 0 & 0 \\ \vec{B}_2 x_1 x_2 & \vec{B}_{n+2} x_2^2 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \vec{B}_n x_1 x_n & \vec{B}_{2n} x_2 x_n & \cdots & \vec{B}_{\frac{n^2+3n-2}{2}} x_n^2 & 0 \\ \vec{B}_{n+1} x_1 & \vec{B}_{2n+1} x_2 & \cdots & \vec{B}_{\frac{n^2+3n}{2}} x_n & \vec{B}_{\frac{(n+1)(n+2)}{2}} \end{bmatrix}\right)\right) - sum\left(\begin{bmatrix} b_1(x_1 + x0_1)L_1 \\ \vdots \\ b_n(x_n + x0_n)L_n \\ b_{n+1} \end{bmatrix}\right) \\
&= sum\left(sum\left(\begin{bmatrix} \vec{B}_1 x_1^2 & 0 & \cdots & 0 & 0 \\ \vec{B}_2 x_1 x_2 & \vec{B}_{n+2} x_2^2 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \vec{B}_n x_1 x_n & \vec{B}_{2n} x_2 x_n & \cdots & \vec{B}_{\frac{n^2+3n-2}{2}} x_n^2 & 0 \\ (\vec{B}_{n+1} - b_1 L_1) x_1 & (\vec{B}_{2n+1} - b_2 L_2) x_2 & \cdots & (\vec{B}_{\frac{n^2+3n}{2}} - b_n L_n) x_n & \left(\begin{array}{c} \vec{B}_{\frac{(n+1)(n+2)}{2}} - \\ (b_{n+1} + \sum_{i=1}^{n} b_i x0_i L_i) \end{array}\right) \end{bmatrix}\right.\right. \\
&\equiv sum(sum(eB \cdot * u \cdot u'));
\end{aligned}$$

Lines $96-106$ evaluate the error function for all grid points. Each grid point is shared by 2^n cubes, however, the quadratic polynomial are different thus the error function is not continuous over cubes. Therefore, the error should be computed for all cubes that share the grid point. For the splitted cubes, we do not have this problem. Lines $99-100$ compute the indices with a similar way in the old method. And lines $104-106$ evaluate the error function $u' \cdot eA \cdot u$

Then, we enter the main loop. Lines $114-116$ evaluate the error function on the center point of all cubes, which provide a lower bound. Lines $120-121$ compute an upper bound using the Lipschitz method. Then the lower and upper bounds are compared. If the gap is smaller than the tolerance, we return the upper bound of error.

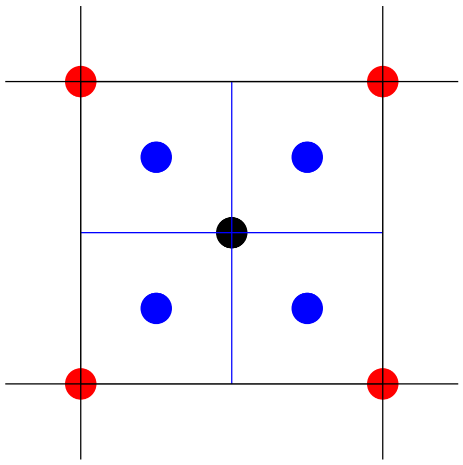


Figure 1.11: Evaluation of error function on grid points

Otherwise, we split the cube into smaller one in lines $133 - 137$. Lines $133 - 134$ generates the lower bound for all 2^n subcubes, and line 135 add the length of cube to generate upper bounds. Here, the indices from line 100 is reused because they have the same pattern. Other variables are duplicated for the next iteration.

However, the number of cubes increase dramatically (8x for 3D and 16x for 4D). And it is not necessary to split all cubes. If the upper bound error of a cube is still smaller than the total lower bound, the cube can be dropped. Lines $126 - 130$ implements this feature.

It should be noticed that the center point evaluated in previous iteration is the cube vertices of this iteration. Therefore, the error function does not need to be evaluated again (see line 117). As shown in figure 1.11, each red point is shared by four black squares with different error function, therefore, it should be evaluated four times for all four black squares. However, when the black square is splitted into blue sub squares, the black point is shared by four subsquares with the same error function, thus it can be computed only once. And the black points are evaluated when working on the black squares, therefore, only blue points should be evaluated when working on blue subsquares of the next iteration.

Performance and Experimental Result

We used 5000 random generated bounding box to analyze the performance. A 3D model is used, 4D model have not been tested yet. The tolerance is set to 0.2.

For the result, we find the approximation error (compare with the result from brute-force method) is usually small. The maximum relative error is 1.6%

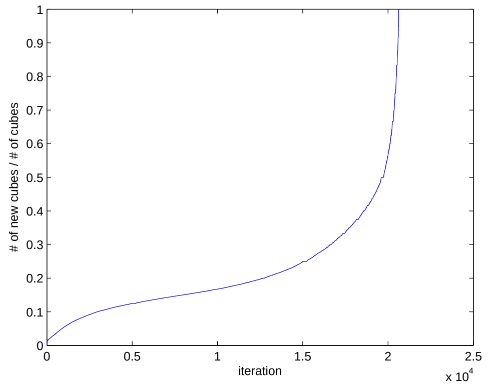


Figure 1.12: Percent of useful cubes for next iteration

and the average is only 0.07%. And only 11 cases with greater than 1% error.

The performance is a little more than 3 times faster. *repmat* function costs 30% of time and *quadBounds* function costs 14%. *sub2ind* function also uses about 9% of total time. The detail is saved in $\sim /Matlab/Profile/interplip$ directory.

The average number of iterations is 5.2 per function[8], which can be reduced by allowing a larger error.

The percent of cubed used in the next iteration is shown in figure 1.12. We can see that the average perecent is 21.68%, the geometry average is 17.54%. There are 14.4% iterations with less than 10% cubes than previous one, 61.41% for less than 20%, 73.48% for less than one fourth, 83.44% for less than one third and 95.14% for less than one half. There are about 25% of iterations with less than 12.5% cubes, given $2^3 = 8$, this means about 25% iterations has less number of subcubes than previous one. With the geometry average, we can see the number of cube is about $(0.1754 \times 8)^{(5.2-1)} = 4$ times larger at the last iteration.

We should change the code if *'out of memory'* error is found.

When we set the tolerance to 0.05, the program is about 3.8 times faster than the brute-force method. With maximum relative error as 3.84% and average error as 0.3%, and less than 1% results has greater than 2% error. The number of iterations per function is reduced to 4.

1.2.4 Linearization Method with LP[9]

[8] This is not correct because of a bug on line 107. We forget to find max of all cubes as the lerr. Therefore, there are more iterations and the gap between lerr and uerr is larger. The value should be around 3. And the gap between lerr and uerr should be smaller.

[9] March 6, 2009

In section 1.2.2 1.2.3, the linearization method is developed and optimized. However, it only works on hyper-cube regions, which might be much larger than necessary. Therefore, we want to add an LP to describe more accurate space. The interface of function is changed to

```
function [b,err] = quadLinfit(model,bbox,lp)
```

where the intersection of lp and bbox should not be empty

We can see that the computation of c vector and err value only works on a set of cubes. How many number of cubes and where these cubes are from do not change the result. Therefore, the only change is to computed M based on cubes similar with c and err. This is simple. We only compute the integral in each cubes and finally computes the sum.

$$\oint_{Lo}^{Hi} \bar{f}(x)x_i \, dX \quad = \quad \sum_{k=1}^{nc} \oint_{Lo_k}^{Hi_k} f(x)x_i \, dX$$

A small difference between the computation of M and c is that the variable of $\bar{f}$ function is the global position where variables of f is normalized. Therefore, the originial value should be restore from the relative value.

The code is

```
oL = (L+S).*repmat(model.dv,1,nc); % original value
oH = (H+S).*repmat(model.dv,1,nc);
diffs = [oH-oL; (oH.^2-oL.^2)/2; (oH.^3-oL.^3)/3];
dind = sub2ind([n,3],repmat(1:n,n+1,n+1),pM+1);
dind = reshape(dind',[],1);
dind = sub2ind([3*n,nc],repmat(dind,1,nc),repmat(1:nc,(n+1)^2*n,1));
M = diffs(dind);
M = prod(reshape(M,n,[]),1); % prod integral
M = sum(reshape(M,[],nc),2); % sum over all cubes
M = reshape(M,n+1,n+1);
```

Now, the problem is how to trim cubes that does not satisfy the LP. We know that *the intersection of lp and a cube is empty iff at least one constraint of lp is not satisfied by all vertices of the cube.* This can be proved as 1) if the intersection is empty, at least one constraint is not satisfied obviously 2) if one constraint is not satisfied, any points in the convex cube does not satisfy the constraint thus the lp. The code is implemented as

```
ox = (x+repmat(S,1,np)).*repmat(model.dv,1,nc*np); % restore the absolute value
ind = lp.A*ox > repmat(lp.b,1,nc*np);
ind = reshape(ind,[],np);
ind = all(ind,2); % all cubes violates a constraint
ind = reshape(ind,[],nc);
ind = ~any(ind,1); % any constraint is not satisfied by all vertices
L = L(:,ind); H = H(:,ind);
S = S(:,ind); Bs = Bs(:,ind);
x = x(:,repmat(ind,1,np));
nc = size(L,2);
```

where x is the vertices for all cubes computed the same way with section 1.2.3.

Of course, we can use the lp to remove more subcubes during the iteration.

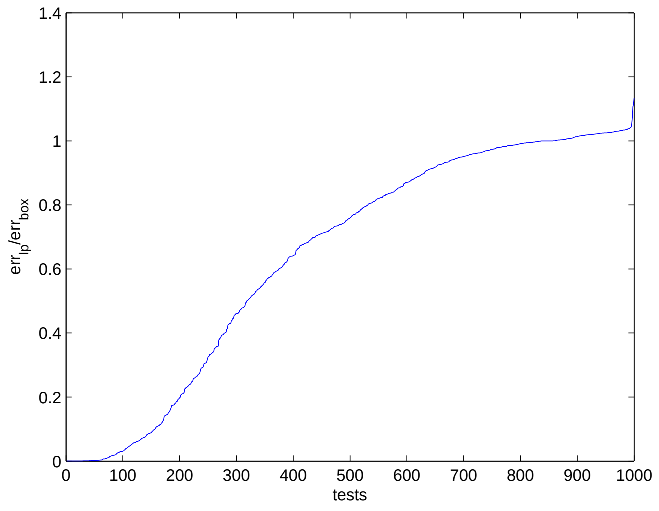


Figure 1.13: Comparsion of errors with/wo LP

Experimental Result

First, we run 1000 random examples for nmos transistor with lp only at the beginning. We compute $\frac{err_{lp}}{err_{bbox}}$ to compare the error with/without lp. The average value is 0.6435. However, the maximum is 1.1344 which is greater than 1. It is reasonable because with lp the cubes used to compute M and c are changed, thus the result of b is different. Our method optimizes the L_2 norm error rather than the L_1 norm error. Thus, the error might be larger with the lp. The percent is about 15%. The detail is shown in figure 1.13 The number of cubes removed by LP is similar. We computed $rs = \frac{cubes_{lp}}{cubes_{bbox}}$. The average value is 0.6039 and the minimum value is 0.0944. For details, see figure 1.14 The performance is a little better with the LP. The value $rt = \frac{time_{lp}}{time_{bbox}}$ is 0.9153 using *cputime* and is 0.9156 from profiler. The average time for each call with lp is 96.8/1000 = 0.097 seconds[10]. The computation time of M matrix rises from 1.5 seconds to about 9.5 seconds. The time spent on solving the lp and remove cubes is about 7.2 seconds (7 % of total time). The average iterations per call is reduced from 2.971 to 2.605.

Then, we add the LP to the inner loop and repeat the experiences (another 1000 random bbox and lp). The average error ratio is 0.5902, the maximum is 1.1031, with 13% greater than 1. The average number of cubes after LP at the beginning is 0.6165 and minimum value is 0.1026. The average number of cubes after LP in the iteration is 0.7926 and minimum value is 0.1250. See figure 1.15 for details. The running time with LP in each iteration is almost the

[10]The profile data is saved in $Profile/linfitQuadFitLip_lp.mat$

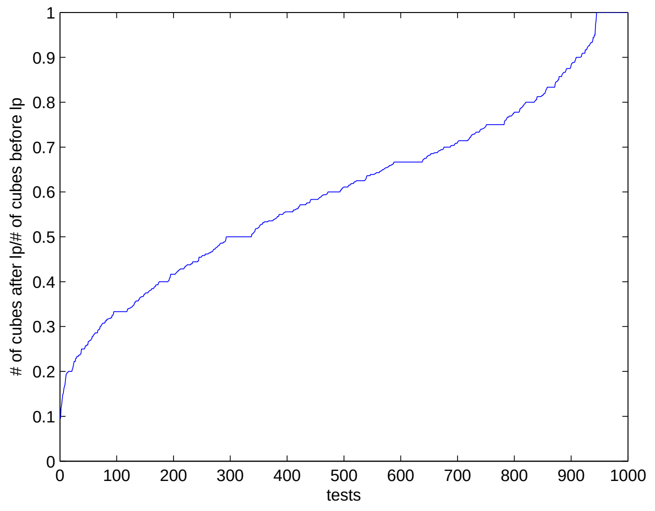


Figure 1.14: Comparsion of number of cubes with/wo LP

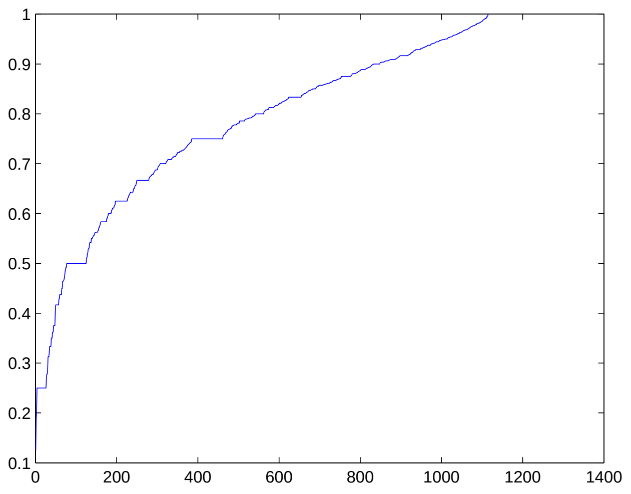


Figure 1.15: Comparsion of number of cubes with/wo LP in Iteration

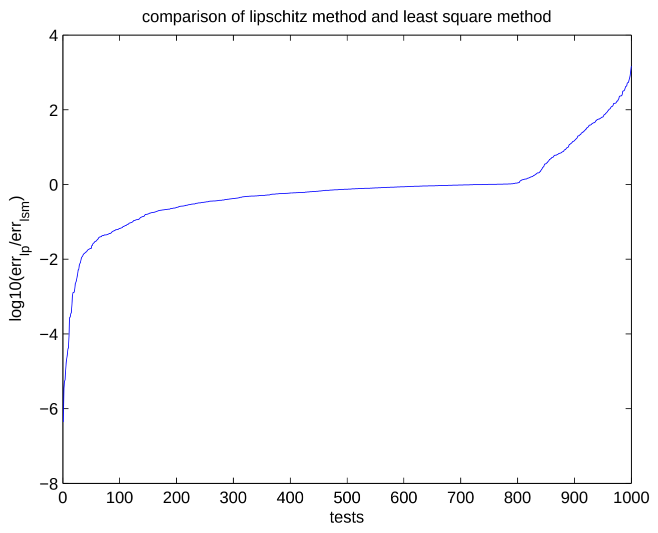


Figure 1.16: Comparsion of errors of lipstichz interpolation and least square method

same with the time of function using bbox only (0.9952). The totoal running time is about 2.7 seconds[11]. The time spent on LP in iterations is about 2.8 seconds (10%). The time spent on LP at the beginning is about 2.1 seconds. The average iteration per call is reduced from 2.950 to 2.265.

Third, we add the LP feature to the brute-force linearization method. We run another 1000 random example compared with brute-force method without LP. The average ratio of eror is 0.6795, the maximum 1.1376 with about 7.4% greater than 1. The proformance is better. The ratio of running time is 0.7842 using *cputime* and 0.7844 by profiler[12]. The total running time is 204.6 seconds for the function with LP. The time spent on LP (also the computation of all vertices) is about 9.3 seconds (4.5%). The computation time of M matrix rises from 1.5 seconds to about 8 seconds.

Finally, we compare the lipschitz method with lp at beginning with the $linfit$. The average ratio of error is 16.9(not useful), the geometry average is 0.7014. the maximum is 1483 with 25.8% greater than 1, and the minimum value is $4.4e-7$. See figure 1.16 for detail. The performance is much better[13]. The ratio of running time is $\frac{43.627}{97.245} = 0.4486$. For the $linfit\_lls$ function, only 4.3% of time is used to compute the coefficient. 41.4% of examples use brute-force method to compute the error using 6.7% of time. The other 58.6% use $linfitErr\_conv$ method with 88.5% of time!

[11]The profile data is save in $Profile/linfitQuadFitLip_iterlp.mat$
[12]The profile is save in $Profile/linfitQuadFit_lp.mat$
[13]saveed in $Profile/linfitQuadFitlip_lpLsm.mat$

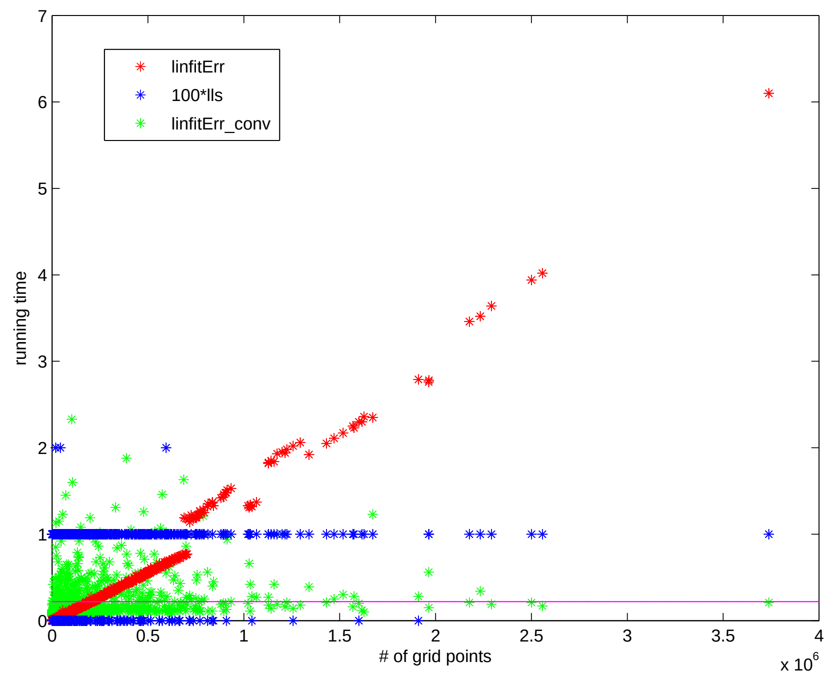


Figure 1.17: Comparsion of running time of different error computation

When the LP feature is disabled (least square method only supports bbox), the geometry average ratio of error is 2.3873, the maximum is 2189 with 48.8% greater than 1, and the minumum is 0.0120. The ratio of running time is about $\frac{46.045}{97.245} = 0.4735$.

From the result, I want to find the best threshold between $linfitErr$ and $linfitErr\_conv$. I found that $2e5$ is a better solution by comparing their running time. See figure 1.17 for details. The magenta line is the average running time of $linfitErr\_conv$ function. The intersection with red line is around $2e5$.

With the new threshhold, the ratio of running time is $\frac{26.7}{50.4} = 0.5298$ for lipschitz method with lp and is $\frac{26.5}{50.4} = 0.5258$ for lipschitz without lp. While 65% of examples use brute-force method with 9.5 seconds, and 35% of examples use convex methods with 37.8 seconds.

1.3 Linearization of Input

In Coho, the input is specified by an brockett's annulus. And annulus is bounded by two ellipses

$$\frac{(x-x0_o)^2}{a_o^2}+\frac{y^2}{b_o^2} = 1 \tag{1.36}$$
$$\frac{(x-x0_i)^2}{a_i^2}+\frac{y^2}{b_i^2} = 1 \tag{1.37}$$

The problem is to find a linear approxmation $y \in px+q \pm u$ for $x \in [x_l, x_h]$.

The first two methods overapproximates the ellipse by polygon and linearizes the polygon.

1.3.1 LP method

The problem is an optimization problem

$$\begin{aligned}
& minu && (1.38)\\
s.t. & && (1.39)\\
px+q+u &\geq \text{all polygon vertices} && (1.40)\\
px+q-u &\leq \text{all polygon vertices} && (1.41)\\
x &\leq x_h && (1.42)\\
x &\geq x_l && (1.43)
\end{aligned}$$

The function is implemented in `mrg_fit` function.

1.3.2 Least Square Method

Given the interval of x, we can compute its upper hull and lower hull. The line in the middle of upper/lower hull can be computed its best linear fit can be computed easily by `polyfit` function in matlab.

1.3.3 Least Square Method of Ellipse

However, the computation of upper and lower hull is error-prone[14]. We can use the ellipse directly which can reduce the overapproximation error and also improve performance although the function is not the bottleneck.

[14] We have to know which brockett region to work on. Thus the check of bound may produce error in matlab.

Given a value x, the y value on outer/inner ellipse is

$$\begin{aligned}
y_o &= \frac{b_o}{a_o}\sqrt{a_o^2 - (x - x0_o)^2} \\
y_i &= \frac{b_i}{a_i}\sqrt{a_i^2 - (x - x0_i)^2} \\
y_m &= \frac{y_i + y_o}{2} \\
\hat{y} &= px + q
\end{aligned}$$

The least square method is

$$\begin{aligned}
minE &= \int_{x_l}^{x_h} (\hat{y} - y_m)^2 \, dx \\
\frac{dE}{dp} &= 2\int_{x_l}^{x_h} (\hat{y} - y_m)x \, dx \\
\frac{dE}{dq} &= 2\int_{x_l}^{x_h} (\hat{y} - y_m) \, dx
\end{aligned}$$

Let $\frac{dE}{dp} = \frac{dE}{dq} = 0$, we have

$$\begin{aligned}
\int_{x_l}^{x_h} \hat{y}x\,dx &= \int_{x_l}^{x_h} y_m x\,dx \\
\int_{x_l}^{x_h} \hat{y}\,dx &= \int_{x_l}^{x_h} y_m\,dx \\
\int_{x_l}^{x_h} \hat{y}x\,dx &= \int_{x_l}^{x_h} (px+q)x\,dx \\
&= \frac{1}{3}x^3|_{x_l}^{x_h} \cdot p + \frac{1}{2}x^2|_{x_l}^{x_h} \cdot q \\
\int_{x_l}^{x_h} \hat{y}\,dx &= \frac{1}{2}x^2|_{x_l}^{x_h} \cdot p + x|_{x_l}^{x_h} \cdot q \\
\int_{x_l}^{x_h} y\,dx &= \int_{x_l}^{x_h} \frac{b}{a}\sqrt{a^2-(x-x0)^2}\,dx \\
&= \frac{b}{a}\int_{x_l-x0}^{x_h-x0} \sqrt{a^2-x^2}\,dx \\
&= \frac{b}{2a}\left(x\sqrt{a^2-x^2} + a^2\arcsin\frac{x}{a}\right)\Bigg|_{x_l-x0}^{x_h-x0} \\
\int_{x_l}^{x_h} yx\,dx &= \int_{x_l}^{x_h} \frac{b}{a}\sqrt{a^2-(x-x0)^2}x\,dx \\
&= \frac{b}{a}\int_{x_l-x0}^{x_h-x0} \sqrt{a^2-x^2}(x+x0)\,dx \\
&= \frac{b}{a}\left(\int_{x_l-x0}^{x_h-x0} \sqrt{a^2-x^2}x\,dx + x0\int_{x_l-x0}^{x_h-x0} \sqrt{a^2-x^2}\,dx\right) \\
&= -\frac{b}{3a}(a^2-x^2)^{\frac{3}{2}}\Bigg|_{x_l-x0}^{x_h-x0} + x0\int_{x_l}^{x_h} y\,dx \\
&= \left(-\frac{b}{3a}(a^2-x^2)^{\frac{3}{2}} + \frac{bx0}{2a}\left(x\sqrt{a^2-x^2} + a^2\arcsin\frac{x}{a}\right)\right)\Bigg|_{x_l-x0}^{x_h-x0} \\
\int_{x_l}^{x_h} y_m\,dx &= \frac{1}{2}\left(\int_{x_l}^{x_h} y_o\,dx + \int_{x_l}^{x_h} y_i\,dx\right) \\
\int_{x_l}^{x_h} y_m x\,dx &= \frac{1}{2}\left(\int_{x_l}^{x_h} y_o x\,dx + \int_{x_l}^{x_h} y_i x\,dx\right)
\end{aligned}$$

Thus, p, q can be solved by linear system.

$$\left[\begin{array}{c} p \\ q \end{array}\right] = \left[\begin{array}{cc} \frac{1}{3}x^3|_{x_l}^{x_h} & \frac{1}{2}x^2|_{x_l}^{x_h} \\ \frac{1}{2}x^2|_{x_l}^{x_h} & x|_{x_l}^{x_h} \end{array}\right] \backslash \left[\begin{array}{c} \int_{x_l}^{x_h} y_m x\,dx \\ \int_{x_l}^{x_h} y_m\,dx \end{array}\right] \tag{1.44}$$

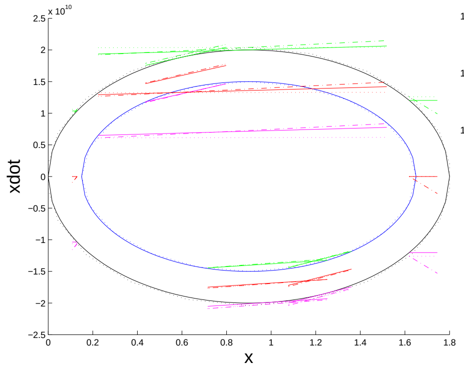
Then, find the extreme value of $e = \hat{y} - y$

$$\begin{aligned}
\frac{de}{dx} &= 0 \\
\frac{d\hat{y}}{x} &= \frac{dy}{dx} \\
p &= -\frac{b}{a}\frac{x - x0}{\sqrt{a^2 - (x - x0)^2}} \\
ap\sqrt{a^2 - (x - x0)^2} &= -b(x - x0) \\
a^2p^2(a^2 - (x - x0)^2) &= b^2(x - x0)^2 \\
a^4p^2 &= (a^2p^2 + b^2)(x - x0)^2 \\
x &= x0 \pm \frac{a^2p}{\sqrt{a^2p^2 + b^2}}
\end{aligned}$$

When $x \geq x0$, $q \leq 0$, thus, $x = x0 - \frac{a^2p}{\sqrt{a^2p^2+b^2}}$. We should find the maximum e_i, e_o over $[x, x_l, x_h]$.

1.3.4 Experience

Figure 1.18 shows the result of three methods. The solid line is for the ellipse method, the dashed line is for the LP method and the dotted line is for the least square method. We can see that the result is quite similar. However, the LP method has strange result on stable state. This is because there are multiple optimal points, and LP solver seems always return the 'bad' one.



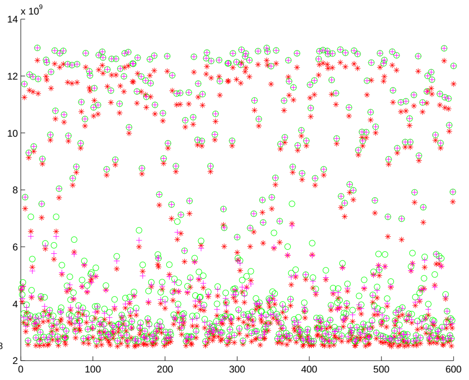


Figure 1.18: Result of different methodsFigure 1.19: Error of different methods

Figure 1.19 compares the error from three methods. The red one is for the ellipse method, green one for least square method and magenta one for LP methods. The average error is

method	**ellipse**	**lsf**	**LP**
arithmetic	5.7066e9 (1)	6.0117e9 (1.05)	5.9529e9 (1.04)
geometry	4.7998e9 (1)	5.1068e9 (1.06)	5.0373e9 (1.05)

1.4 First Order Model

1.4.1 Linear approximation of polynomial function

By Taylor's theorem, an function can be approximated as

$$\tilde{f}(x) \approx f(x_0) + J(x_0) \cdot (x - x_0)$$

where J is the *Jacobian Matrix*.

For second order polynomial function

$$\begin{aligned} f(x) &= u'Au \\ u &= [x;1] \\ A' &= A \\ J(x) &= 2(Au)' \\ &= 2u'A \end{aligned}$$

the approximation is

$$\begin{aligned} \tilde{f}(x) &\approx u_0'Au_0 + 2u_0'A(u - u_0) \\ &= u_0'A(2u - u_0) \end{aligned}$$

the error term is

$$\begin{aligned} \Delta f(x) &= f - \tilde{f} \\ &= u'Au - 2u_0'Au + u_0'Au_0 \\ &\equiv u'Au - 2\mathbf{b}'u + \mathbf{c} \\ &= u'Bu \\ B &= \begin{pmatrix} a_{11} & \cdots & a_{1n} & a_{10} - \mathbf{b_1} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nn} & a_{n0} - \mathbf{b_n} \\ a_{01} - \mathbf{b_1} & \cdots & a_{0n} - \mathbf{b_n} & a_{00} - \mathbf{2b_0} + \mathbf{c} \end{pmatrix} \end{aligned}$$

If the feasible region is a cube. The extreme value locates at critical points where derivative is zero or boundary.

$$\begin{aligned} \frac{d\Delta f}{du} &= 2Bu \\ \frac{d\Delta f}{dx} &= 2B(1:n,:)u \end{aligned}$$

Then the stationary point is

$$A(1:n, 1:n) \cdot x \quad = \quad b - A(1:n, end)$$

To find the boundary, it is similar. For example, if we want to find the extreme value on $x_k = v_k$ face, then we should discard $\frac{d\Delta f}{dx_k} = 0$ and add $x_k = v_k$ constraint to the linear system. Then, all critical points can be find by solving 3^n linear systems for a second order polynomial function in a cube.

If the feasible region is a convex region described by linear constraints, it is qudratic programming problem. We can use *quadpoly* or *CPLEX* to solve it.

First order model

The current function is

$$\begin{aligned}
u &= [v_s; v_g; v_d; 1] \\
\frac{f(v_s, v_g, v_d)}{k} &= \begin{cases} u'A_c u & v_g - v_s - v_t \le 0 \\ u'A_l u & v_g - v_s - v_t \ge 0, v_g - v_d - v_t \ge 0 \\ u'A_s u & v_g - v_s - v_t \ge 0, v_g - v_d - v_t \le 0 \end{cases} \\
A_c &= \frac{1}{2}\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \\
A_l &= \frac{1}{2}\begin{pmatrix} 1 & -1 & 0 & 1 \\ -1 & 0 & 1 & 0 \\ 0 & 1 & -1 & -1 \\ 1 & 0 & -1 & 0 \end{pmatrix} \\
A_s &= \frac{1}{2}\begin{pmatrix} 1 & -1 & 0 & 1 \\ -1 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 1 \end{pmatrix}
\end{aligned}$$

Thus, given any cube contained by only one region, the linear approximation and error bound can be computed directly. The region is shown in figure 1.20. If the cube intersects more than one region, then we use the linear approximation from the region with most intersection volumns. And for the error computation, it is similar

$$\begin{aligned}
\Delta f(x) &= f - \tilde{f} \\
&= u'Au - 2u_0'\hat{A}u + u_0'\hat{A}u_0 \\
\frac{\Delta f(x)}{du} &= 2Au - 2u_0'\hat{A}
\end{aligned}$$

It is about 8 times faster than the interp method.

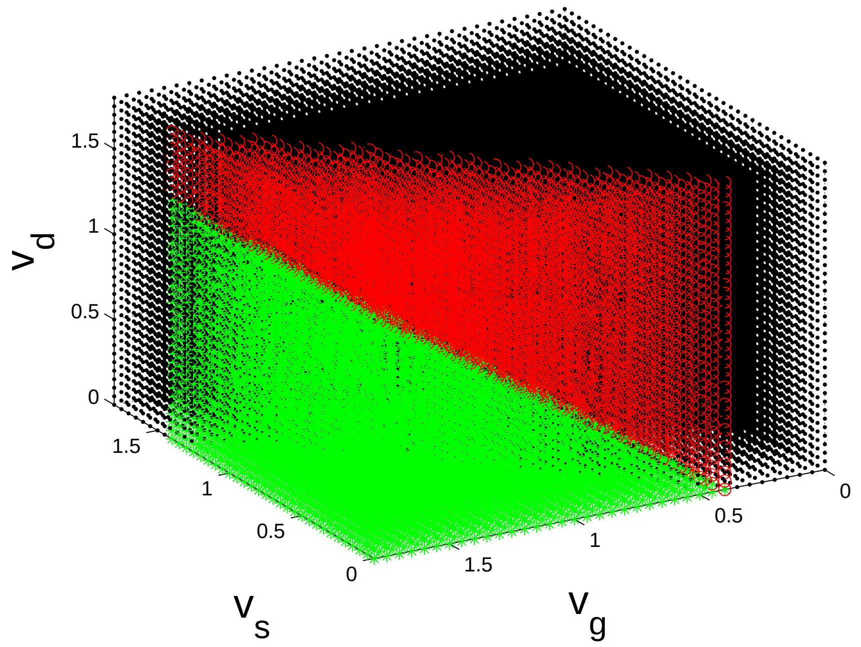


Figure 1.20: The region of ideal transistor