Discrete Adjoint for TITAN2D

H. AGHAKHAI

Contents

The aim of this report is to briefly show how the discrete adjoint for the system of Savage_Hutter partial differential equations is computed.

1 Mathematical Definition

1.1 Savage_Hutter Equation

To simulate granular pyroelastic flow, resulted from a volcanic avalanche TITAN2D solves Savage_Hutter Equation which is a modified version of Shallow Water (SW) equation presented in 1989 [2].

$$U_t + F(U)_x + G(U)_y = S(U) \tag{1}$$

Where:

$$U = (h, hv_x, hv_y)^T$$
$$F = (hv_x, hv_x^2 + 0.5k_{ap}g_z h^2, hv_x hv_y)^T$$
$$G = (hv_y, hv_x v_y, hv_y^2 + 0.5k_{ap}g_z h^2)^T$$
$$S = (0, S_x, S_y)^T$$
$$S_x = g_x h - \frac{V_x}{\sqrt{V_x^2 + V_y^2}}\left(g_z h + \frac{hV_x^2}{r_x}\right)\tan(\phi_{bed}) - hk_{ap}\mathrm{sgn}\left(\frac{\partial V_x}{\partial y}\right)\frac{\partial(g_z h)}{\partial y}\sin(\phi_{int})$$
$$S_y = g_y h - \frac{V_y}{\sqrt{V_x^2 + V_y^2}}\left(g_z h + \frac{hV_y^2}{r_y}\right)\tan(\phi_{bed}) - hk_{ap}\mathrm{sgn}\left(\frac{\partial V_y}{\partial x}\right)\frac{\partial(g_z h)}{\partial x}\sin(\phi_{int})$$

1.2 Adjoint definition and formulation

Let U and V be two vector spaces, and L be a linear operator that maps any $u \in U$ into $v \in V$. And $< \cdot, \cdot >$ be a bilinear map that maps any two vectors like u, v two a real number, $U \times V \to \mathbb{R}$. Then the adjoint operator, L^*, of L is defined:$< Lu, v >=< u, L^* v >$.
Given $R(U, \alpha)$ as a system of governing equations, where U is the solution vector, and α is the vector of design parameters.
The object is to minimize $\quad \mathrm{J}(U, \alpha) \quad$ subject to $\quad R(U, \alpha) = 0$
So in optimization context we can say that we want to optimize the goal functional under the restriction of the governing equations. If we write the first variation of the functional and the governing equations for a discrete set of points, we will have:

$$\frac{dJ}{d\alpha} = \frac{\partial J}{\partial U}\frac{dU}{d\alpha} + \frac{\partial J}{\partial \alpha} \tag{2}$$

and:

$$\frac{\partial R}{\partial U}\frac{dU}{d\alpha} + \frac{\partial R}{\partial \alpha} = 0 \tag{3}$$

replacing $\frac{dU}{d\alpha}$ from the second equation into the first equation leads to:

$$\frac{dJ}{d\alpha} = -\frac{\partial J}{\partial U}(\frac{\partial R}{\partial U})^{-1}\frac{\partial R}{\partial \alpha} + \frac{\partial J}{\partial \alpha} \tag{4}$$

Previous equation shows that we can compute the sensitivity in two different ways:

1. Forward mode: first computes $(\frac{\partial R}{\partial U})^{-1}\frac{\partial R}{\partial \alpha}$
2. Adjoint mode: first computes $\frac{\partial J}{\partial U}(\frac{\partial R}{\partial U})^{-1}$

In the adjoint mode the gradient of functional is obtained by a forward solution of U, and one backward solution for the adjoint, and it is independent from α. Thus if the number of design parameters be greater than the number of objective functions, then the computational cost of the adjoint method is much lower than the forward method. To connect the above formulation with the adjoint concept, we can write:

$$\begin{aligned} u = \frac{dU}{d\alpha}, \qquad A &= \frac{\partial R}{\partial U} \\ g^T = \frac{\partial J}{\partial U}, \qquad f &= -\frac{\partial R}{\partial \alpha} \end{aligned} \tag{5}$$

Forward method:

$$\begin{aligned} \frac{dJ}{d\alpha} &= g^T u + \frac{\partial J}{\partial \alpha} \\ \text{Subject to} \quad Au &= f \end{aligned} \tag{6}$$

Adjoint Method:

$$\begin{aligned} \frac{dJ}{d\alpha} &= v^T f + \frac{\partial J}{\partial \alpha} \\ \text{Subject to} \quad A^T v &= g \end{aligned} \tag{7}$$

From the above we can see $< Au, v >=< u, A^T v >$,where v is the adjoint vector and is the solution of the following system of equations:

$$(\frac{\partial R}{\partial U})^T v = (\frac{\partial J}{\partial U})^T \tag{8}$$

2 Adjoint Computation

As shown in equation 8 the adjoint equation is:

$$(\frac{\partial R}{\partial U})^T v = (\frac{\partial J}{\partial U})^T$$

TITAN2D solves hyperbolic system of equations of Savege_Hutter, using Godunov scheme finite volume with HLL solver to compute the flux terms. The discritized form of the equations can be written:

$$U_i^{n+1} = U_i^n - \frac{\triangle t}{\triangle x}\{F^n_{i+\frac{1}{2}} - F^n_{i-\frac{1}{2}}\} - \frac{\triangle t}{\triangle y}\{G^n_{i+\frac{1}{2}} - G^n_{i-\frac{1}{2}}\} \tag{9}$$

$(\frac{\partial R}{\partial U})^T_{m\times m} = K_{ij}$ which m is the number of time steps

every K_{ij} is also a $n \times n$ matrix, which n is the number of elements. For the Godunov method:

$$K_{i,i} = I \quad \text{and} \quad K_{i,i+1} = (\frac{\partial R_p^{i+1}}{\partial U_q^i})^T \quad \& \quad \text{rest the of elements} = 0$$

The 2nd term is the sensitivity of the residual vector in time step (i+1) with respect to the state variables in time step i, that has to be evaluated at element p with respect to element q, consequently:

$$(\frac{\partial R}{\partial U})^T_{m\times m} = \begin{pmatrix} I & K_{1,2} & & & \\ & I & K_{2,3} & 0 & \\ & & \ddots & \ddots & \\ & 0 & & I & K_{m-1,m} \\ & & & & I \end{pmatrix} \tag{10}$$

Taking to account that in numerical methods the solution at each point only depends on the neighbor points so every $K_{i,i+1}$ is also a block banded matrix. We used first order derivative so the state variables in each elements only depends on its neighbors.

$$\begin{aligned} v_1 + K_{1,2}v_2 &= \left(\frac{\partial J}{\partial U}\right)^T_1 \\ &\vdots \\ v_{m-1} + K_{m-1,m}v_m &= \left(\frac{\partial J}{\partial U}\right)^T_{m-1} \\ v_m &= \left(\frac{\partial J}{\partial U}\right)^T_m \end{aligned} \tag{11}$$

The previous equation means that to compute the Jacobian matrices, the solution vectors for all of the elements and for all of time steps have to be stored. Then we can compute the adjoint in a reverse time order. Since it is impossible to store all of these matrices in the memory, people use dynamic check pointing schemes and appropriate parallel I/O to overcome these difficulties.

3 Error Estimation

Suppose that $J(Q)$, $J(Q_h)$ and $J(Q_H)$ are respectively exact value, numerical value on fine mesh, and numerical value on a coarse mesh of objective functional. We want to minimize the numerical error of objective functional $|J(Q)-J(Q_H)|$. Following the steps described in [1] we can write:

$$J(Q_h) \approx J(Q_h^H) - \underbrace{(\psi_h^H)^T R(Q_h^H)}_{\text{Adjoint correction term}} - \underbrace{(\psi_h - \psi_h^H)^T R(Q_h^H)}_{\text{Remaining term}}$$

where Q_h^H and ψ_h^H are reconstruction of the flow and adjoint solution from coarse mesh to embedded mesh, we use linear reconstruction to approximate ψ_h

and Q_h, and use constant reconstruction to approximate ψ_h^H and show them respectively with ψ_L and ψ_C, so the above equation changes to

$$J(Q_h) \approx J(Q_L) - (\psi_L)^T R(Q_L) - (\psi_L - \psi_C)^T R(Q_L) \tag{12}$$

So the total error is going to be

$$e_k = \sum |(\psi_L - \psi_C)^T R(Q_L)| \tag{13}$$

Given a threshold for the functional, TOL, so the local error parameter is going to be

$$\frac{TOL}{N} \tag{14}$$

then we can normalize the error in the element by

$$r_k = \frac{e_k}{t} \tag{15}$$

4 Computer Programming

4.1 General algorithm

As discussed above in the theroy section, there are two important data that we need to compute discrete adjoint for TITAN2D, or possibly any other code that uses Euler explicit, which are the functional sensitivity with respect to the particular element at specific time step and the jacobian matrix. We can compute the functional sensitivity in forward run, it is also possible to compute the jacobian in forward run, but it is not a good idea. Because in that can one has to store all jacobians for all time steps. Instead of this, we can just store the solution and compute the jacobian, matrix in reverse run whenever we want. This strategy not only helps to avoid storing all jacobian matrices, but also allows to just have one jacobian matrix and clear it after computing the compounding adjoint vector. Moreover storing solution requires very very less required memory instead of storing Jacobian matrices. In the next part we discuss about the data structures that we create to compute Jacobain, and here we just talk about the general algorithm.

Given the required data to compute the adjoint the general algorithm is as follows. At the end of forward run dual_solver function is called in hpfem and inside the main function. This function calls the other functions we need to compute the adjoint. As showed before for the last time step of forward run or first time step of reveres run we do not need to compute Jacbian and the adjoint vector is simply the functional sensitivity. As also discussed in detail in theory section, beside adjoint computation we also compute the dual weighted error. To compute the error, we first uniformly refine compute the residual, then uniformly unrefine to get back the original grid and then compute the error given the adjoint solution and obtained residual from this procedure. This method is also applied for all time steps, and here we will not explain in further. For computing the adjoint for the next steps a new loop starts. Inside of this loop we first we reverse the state variables. This means that the state variables change with respect to the time step that we want to compute the adjoint. The

we call setup_geoflow function that computes the gravity derivative of gravity and calculates the other required topographic based on the replaced state variables. Next step is to compute the jacobian. Jacobian is simply the sensitivity of the residual with respect to state variables. Since in discrete form the residual of an element depends also to state variables of the neighbor element, we need also to compute the sensitivity of the residual vector of each element with respect to all neighbors that may affect it, and for state variables of the element itself. To compute the jacobian we use forward difference. The general idea is to compute each component we perturb the corresponding state variables and find the change in the residual vector the simply use the definition of the derivative to compute the jacobian. The perturbation that we used in this study is flexible and can be changed, but we mostly tried with $1E-08$, and obtained very good results. After perturbing any state variable, fluxes and slopes are updated and residual function is called to compute the change in residual and compute the jacoboian from equation 16. It is important to return everything back to before the perturbing the state varibles, otherwise it will affect the other computations.

$$jac_{i,j} = \frac{R_i(u_j + h) - R_i(u_j)}{h} \tag{16}$$

$$jac_{i,j} = \frac{1}{2}\left(\frac{R_i(u_j + h) - R_i(u_j)}{h} + \frac{R_i(u_j) - R_i(u_j - h)}{h}\right) \tag{17}$$

Our experienced showed that for some of the element just forward difference is not enough accurate to compute the jacobian and more accurate scheme is required. For these cases we used central difference scheme. To do that, after computing forward difference we compute backward difference and then compute the average of them to find the central difference derivative (equation 17). For element that are located in the boundary the jacobian is zero because regardless of any change on state variables of themselves or their neighbors their value is fix and does not change.
After computing the jacobian for all of the elements, we can now compute the adjoint by calling calc_adjoint function. Then we compute the residual and dual weighted error as we discussed earlier, and go to the next iteration in the loop until we reach the first step of the forward run, or the last step of the reverse run, and during this loop we can call tecplot function to report the results into an ascii file.

4.2 Data structures without AMR

For sake of simplicity and for the first phase We implemented the above computations without Adaptive Mesh Refinement (AMR). For computing the adjoint vector, we created two new Jacobian and Solution classes. Solution class stores the all required information for computing adjoint in reverse run including the vector of solutions. Jacobian class is a data frame that we need to compute the adjoint in reverse run. More clearly, in forward run at each time step and for each element we create a new solution object to store the solution vector and sensitivity of the functional of interest with respect to the current solution for the specific element. To access to this solution object, we create a vector of solution pointers inside the jacobian class that holds the address of this object. The Jacobian class that is created for each element once, also holds the jacobian

matrix of this element. In without refinement code, grid is fix, so each element has always four neighbors. As mentioned before the Jacobian matrix of whole problem is a block bounded matrix which size of this block depends on stencil that we use to compute the residual vector. In TITAN2D, we use central difference scheme to compute the derivatives and HLL to compute the fluxes, so the residual vector of each element only depends on its four neighbor elements and element itself. Consequently, In Jacobian class we create a 3D matrix with size of $5 \times 3 \times 3$. First index in this matrix shows neighbor number, second index shows the residual vector and third index shows the state variable that this component of Jacobian matrix is computed. We numbered the neighbor elements in following format. Neighbor element in positive x side is 1, in positive y side is 2, in negative x side is 3, in negative y side is 4, and for element itself is 0. The order of residual vector and state variables are same as other parts of code. For example, the component of Jacobian matrix related to effect of x momentum of the negative y neighbor element on the residual of y momentum is equal to jacobian[4][2][1], because the neighbor number is 4, residual is 3rd component of residual vector so base on C numbering that starts from zero its number is 2, and the state variable number is 1 because it is the 2nd component of state variables.

To allocate the memory as lowest as possible, we allocate the memory for the jacobian matrix inside the jacobian class exactly when we want it in reverse run, so with this manner we not only keep the jacobian of the whole problem in a matrix free fashion, but also allocate the memory when we need it not at the start of the simulation. With this strategy we will have much more space to store the solution and avoid writing the solution history to disk. In addition to the jacobian matrix and vector of pointers of solution history, we have the required methods in Jacobian class to access to the solution and also the functional sensitivity. These methods are used in reverse run to compute the adjoint. Other data and methods of Solution and Jacobian class can be find in the appendix section.

4.3 Data structures with AMR

Appendices

A Solution class

A.1 Solution header

```
class Solution {

public:

  Solution(double* curr_sol, double kactxy, double* funcsensitivity
    );

  double* get_solution(void);

  double get_kact(void);

  double* get_funcsens(void);

  ~Solution();

protected:

  double funcsens[NUM_STATE_VARS]; //this variable keeps the value
    of sensitivity at each time step for this element.
  double states[NUM_STATE_VARS]; //to save the solution
  double kact; //to save kact

};
```

A.2 Solution source

```
Solution::Solution(double* curr_sol, double kactxy, double*
    funcsensitivity) {

  for (int i = 0; i < NUM_STATE_VARS; ++i)
    states[i] = curr_sol[i];

  kact = kactxy;

  for (int i = 0; i < NUM_STATE_VARS; ++i)
    funcsens[i] = funcsensitivity[i];
}
double* Solution::get_solution() {
  return states;
}

double Solution::get_kact() {
  return kact;
}

double* Solution::get_funcsens() {
  return (funcsens);
}
Solution::~Solution() {

}
```

B Jacobian class

B.1 Jacobian header

```
class Jacobian {
  //friend functions and classes

public:

  //constructors
  Jacobian(unsigned* key, double* position);

  void set_jacobian(int neigh_num, double elemjacob[3], int
    state_vars_num, const double incr);

  // this function sets the jacobian for a boundary element
  void set_jacobian();

  void print_jacobian(int iter);

  double*** get_jacobian(void);

  double* get_solution(void);

  double* get_kact(void);

  void new_jacobianMat(void);

  double* get_funcsens(int iter);

  void put_solution(Solution* sol);

  virtual void rev_state_vars(void* element, int iter);

  void set_jacobianMat_zero(int jacmatind);

  void add_state_func_sens(double* func_sens_prev, int iter);

  void del_jacobianMat();

  double* get_position();

  unsigned* get_key();

  //destructor
  virtual ~Jacobian();

  //members
protected:

  vector<Solution*> solvector;
  unsigned key[DIMENSION];
  double position[DIMENSION];
  double ***jacobianMat; //double jacobianMat [5][3][3], self[3][3],
    neigh1[3][3], neigh2[3][3], neigh3[3][3], neigh4[3][3]

};
```

B.2 Jacobian source

```
Jacobian::Jacobian(unsigned* key, double* position) {
  for (int i = 0; i < 2; ++i) {
    Jacobian::key[i] = key[i];
    Jacobian::position[i] = position[i];
  }

  jacobianMat = NULL;
}

void Jacobian::new_jacobianMat() //in forward run we just save the
    solution and in backward run we compute the jacobian
{
  int i, j, k;
  jacobianMat = new double**[5];
  for (i = 0; i < 5; i++) {
    jacobianMat[i] = new double*[3];
    for (j = 0; j < 3; j++)
      jacobianMat[i][j] = new double[3];
  }

  for (i = 0; i < 5; i++)
    for (j = 0; j < 3; j++)
      for (k = 0; k < 3; k++)
        jacobianMat[i][j][k] = 0.0;

  return;
}

double* Jacobian::get_position() {
  return position;
}

void Jacobian::del_jacobianMat() {

  if (jacobianMat != NULL) {

    for (int i = 0; i < 5; ++i) {
      for (int j = 0; j < 3; ++j)
        delete[] jacobianMat[i][j];

      delete[] jacobianMat[i];
    }
    delete[] jacobianMat;
  }

}

void Jacobian::set_jacobian(int neigh_num, double elemjacob[3], int
     state_vars_num,
    const double incr) {

  int i, j;

  if (state_vars_num < 1) //since state_vars=1 is for first
    component of adjoint
```

```
    i = state_vars_num;
  else
    i = state_vars_num - 1;

  for (j = 0; j < 3; j++)
    jacobianMat[neigh_num][i][j] = elemjacob[j] / incr;

  return;
}

void Jacobian::set_jacobian() {

  for (int i = 0; i < 5; i++)
    for (int j = 3; j < 3; j++)
      for (int k = 0; k < 3; k++)
        jacobianMat[i][j][k] = 0.0;

  return;
}

double*** Jacobian::get_jacobian() {
  return jacobianMat;
}

void Jacobian::print_jacobian(int iter) {

  cout << "iter:  " << iter << '\n';
  cout << "key1:  " << key[0] << "  key2:  " << key[1] << '\n';
  cout << "X:  " << position[0] << "  Y:  " << position[1] << '\n';
  cout << "Jacobian: " << '\n';
  //cout << "self"<<"
  for (int i = 0; i < 5; i++) {
    cout << "Matrix=  " << i << "," << '\n';

    for (int j = 0; j < 3; j++) {
      for (int k = 0; k < 3; k++) {
        cout << scientific << setw(10) << setprecision(8) <<
    jacobianMat[i][j][k] << "  ";
        if (dabs(jacobianMat[i][j][k]) > 10.)
          cout << "Jedi begir mano" << endl;
      }
      cout << '\n';
    }
  }
  return;
}

void Jacobian::put_solution(Solution* sol) {
  solvector.push_back(sol);

  return;
}

void Jacobian::rev_state_vars(void* elementin, int iter) {

  double *state_vars, *prev_state_vars, *kactxy;
  Element* element;
  element = (Element*) elementin;

  state_vars = element->get_state_vars();
  prev_state_vars = element->get_prev_state_vars();
  kactxy = element->get_kactxy();
```

```
114
   for (int i = 0; i < NUM_STATE_VARS; i++)
116     state_vars[i] = *((solvector.at(iter))->get_solution() + i);

118   for (int i = 0; i < NUM_STATE_VARS; i++)
        prev_state_vars[i] = *((solvector.at(iter - 1))->get_solution()
        + i);
120
   *kactxy = (solvector.at(iter))->get_kact();
122
   for (int i = 0; i < 3; ++i)
124     prev_state_vars[6 + i] = state_vars[6 + i];

126   return;
}
128
double* Jacobian::get_funcsens(int iter) {
130   return (solvector.at(iter)->get_funcsens());
}
132
void Jacobian::set_jacobianMat_zero(int jacmatind) {
134
   for (int j = 0; j < 3; j++)
136     for (int k = 0; k < 3; k++)
        jacobianMat[jacmatind][j][k] = 0.0;
138
   return;
140 }

142 void Jacobian::add_state_func_sens(double* func_sens_prev, int iter
    ) {

144   for (int ind = 0; ind < 3; ind++)
      *(get_funcsens(iter) + ind) += func_sens_prev[ind];
146
   return;
148 }

150 unsigned* Jacobian::get_key() {
   return key;
152 }

154 Jacobian::~Jacobian() {

156   del_jacobianMat();

158   vector<Solution*>::iterator it;
   for (it = solvector.begin(); it != solvector.end(); ++it)
160     delete (*it);

162   solvector.clear();

164 }
```

C Some of the most important functions for computing adjoint

C.1 dual_solver.C

```
1
#ifdef HAVE_CONFIG_H
3 # include <config.h>
#endif
5 #include "../header/hpfem.h"

7 #define DEBUG1

9 #define KEY0     3777862041
#define KEY1     2576980374
11 #define EFFELL   0
#define ITER     187
13 #define J        0

15 void dual_solver(HashTable* El_Table, HashTable* NodeTable,
     vector<Jacobian*>* solHyst, MatProps* matprops_ptr,
17     TimeProps* timeprops_ptr, MapNames *mapname_ptr, PertElemInfo*
     eleminfo) {

19   int myid, numprocs;
   MPI_Comm_rank(MPLCOMM_WORLD, &myid);
21   MPI_Comm_size(MPLCOMM_WORLD, &numprocs);

23   const int rescomp = 1;
   const double increment = INCREMENT;
25   const int maxiter = timeprops_ptr->iter;

27 //  // here we do this because iter in timeprops is such that it is
      one iter more than
//  // actual iteration at the end of forward run, so we have to
     correct that.
29 //  timeprops_ptr->iter = timeprops_ptr->maxiter;

31   double functional = 0.0, dt;

33   int adjiter = 0;

35   int hrs, mins;
   double secs;
37
   allocJacoMat(*solHyst); //this function allocates memory to store
      Jacobian matrices
39   //int unsigned key[2] = { KEY0, KEY1 };

41   calc_adjoint(El_Table, solHyst, maxiter, adjiter, myid);

43   uinform_refine(El_Table, NodeTable, timeprops_ptr, matprops_ptr,
     numprocs,
       myid);
45
   error_compute(El_Table, NodeTable, timeprops_ptr, matprops_ptr,
     maxiter, myid,
47       numprocs);

49   double UNREFINE_TARGET = .01; //dummy value is not used in the
     function
   unrefine(El_Table, NodeTable, UNREFINE_TARGET, myid, numprocs,
     timeprops_ptr,
51       matprops_ptr, rescomp);

53   int tecflag = 2;
```

```
    tecplotter(El_Table, NodeTable, matprops_ptr, timeprops_ptr,
      mapname_ptr,
55        functional, tecflag);

57    tecflag = 1;

59    for (int iter = maxiter; iter > 0; --iter) {

61      timeprops_ptr->iter = iter;
      dt = timeprops_ptr->dt.at(iter - 1);
63      adjiter++;

65      // we need this even for  iter = maxiter because after refine
      and unrefine
      // the state variables are not same as forward run
67      reverse_states(El_Table, solHyst, iter);

69      timeprops_ptr->adjoint_time(iter - 1);

71      setup_geoflow(El_Table, NodeTable, myid, numprocs, matprops_ptr
        ,
            timeprops_ptr);
73
      compute_functional(El_Table, &functional, timeprops_ptr);
75      eleminfo->update_dual_func(functional);

77      calc_jacobian(El_Table, NodeTable, solHyst, matprops_ptr,
      timeprops_ptr,
            mapname_ptr, increment);
79
//      print_jacobian(El_Table, solHyst, iter);
81
      calc_adjoint(El_Table, solHyst, iter, adjiter, myid);
83
      if (eleminfo->iter == iter - 1)
85        fill_pertelem_info(El_Table, solHyst, eleminfo);

87      //for first adjoint iteration there is no need to compute
      Jacobian and adjoint can be computed from the functional
      //sensitivity w.r.t to parameters
89
      uinform_refine(El_Table, NodeTable, timeprops_ptr, matprops_ptr
      , numprocs,
91          myid);

93      error_compute(El_Table, NodeTable, timeprops_ptr, matprops_ptr,
       iter, myid,
            numprocs);
95
      // in dual weighted error estimation if solver performs n step,
       we'll have n+1
97      // solution and n+1 adjoint solution, but we'll have just n
      residual and as a
      // result n error estimate. The point is that at initial step
      (0'th step),
99      // we know the solution from initial condition  so the error of
       0th step is zero,
      // and we have to compute the error for other time steps.
101
      double UNREFINE_TARGET = .01; //dummy value is not used in the
      function
103     unrefine(El_Table, NodeTable, UNREFINE_TARGET, myid, numprocs,
```

```
        timeprops_ptr, matprops_ptr, rescomp);

    if (/*adjiter*/timeprops_ptr->ifadjoint_out() /*|| adjiter == 1
    */)
      tecplotter(El_Table, NodeTable, matprops_ptr, timeprops_ptr,
    mapname_ptr,
          functional, tecflag);

  }

  return;
}

int num_nonzero_elem(HashTable *El_Table) {
  int num = 0;       //myid
  HashEntryPtr currentPtr;
  Element *Curr_El;
  HashEntryPtr *buck = El_Table->getbucketptr();

  for (int i = 0; i < El_Table->get_no_of_buckets(); i++)
    if (*(buck + i)) {
      currentPtr = *(buck + i);
      while (currentPtr) {
        Curr_El = (Element*) (currentPtr->value);
        if (Curr_El->get_adapted_flag() > 0)
          num++;
        currentPtr = currentPtr->next;
      }
    }

  return (num);
}

void initSolRec(HashTable* El_Table, HashTable* NodeTable,
    vector<Jacobian*> *solHyst, TimeProps* timeprops_ptr, int myid)
     {

  HashEntryPtr* buck = El_Table->getbucketptr();
  HashEntryPtr currentPtr;
  Element* Curr_El;
  Jacobian* jacobian;
  double functionalsens[3] = { 0., 0., 0. };
  int num = 0;

  solHyst->reserve(num_nonzero_elem(El_Table));

  for (int i = 0; i < El_Table->get_no_of_buckets(); i++) { // this
     part allocate memory and initialize jacobian matrices inside
    the corresponding Jacobian
    if (*(buck + i)) {
      currentPtr = *(buck + i);
      while (currentPtr) {
        Curr_El = (Element*) (currentPtr->value);
        if (Curr_El->get_adapted_flag() > 0) {

          jacobian = new Jacobian(myid, Curr_El->pass_key(),
              Curr_El->get_coord());
          // at time step 0 we do not compute functional
    sensitivity,
          // we compute the contribution of this time step n
    functional sensitivity on time step 1
```

```
//            compute_funcsens(Curr_El, timeprops_ptr, functionalsens
    );
159           Solution *solution = new Solution(Curr_El->get_state_vars
    (),
                  Curr_El->get_kactxy(), functionalsens);
161           Curr_El->put_sol_rec_ind(num);
              jacobian->put_solution(solution);
163           solHyst->push_back(jacobian);
              num++;
165
            }
167         currentPtr = currentPtr->next;
          }
169     }
    }
171 return;
}
173
void allocJacoMat(vector<Jacobian*> solHyst) {
175
  vector<Jacobian*>::iterator it;
177 for (it = solHyst.begin(); it != solHyst.end(); ++it)
    (*it)->new_jacobianMat();
179
  return;
181 }

183 double tiny_sgn(double num, double tiny) {
  if (dabs(num) < tiny)
185   return 0.;
  else if (num > tiny)
187   return 1.;
  else
189   return -1.;
}
191
void orgSourceSgn(Element* Curr_El, double frictiny, double* orgSgn
    ) {
193
  double* d_state_vars_x = Curr_El->get_d_state_vars();
195 double* d_state_vars_y = d_state_vars_x + NUM_STATE_VARS;
  double* prev_state_vars = Curr_El->get_prev_state_vars();
197 double h_inv;
  double tmp = 0.0;
199 double velocity[2];
  for (int i = 0; i < 2; i++)
201   orgSgn[i] = 0.0;

203 if (prev_state_vars[0] > GEOFLOW_TINY) {

205   velocity[0] = prev_state_vars[2] / prev_state_vars[0];
    velocity[1] = prev_state_vars[3] / prev_state_vars[0];
207
  } else {
209   for (int k = 0; k < DIMENSION; k++)
      velocity[k] = 0.;
211 }

213 if (prev_state_vars[0] > 0.0)
    h_inv = 1. / prev_state_vars[0];
215
```

```
  tmp = h_inv * (d_state_vars_y[2] - velocity[0] * d_state_vars_y
    [0]);
  orgSgn[0] = tiny_sgn(tmp, frictiny);

  tmp = h_inv * (d_state_vars_x[3] - velocity[1] * d_state_vars_x
    [0]);
  orgSgn[1] = tiny_sgn(tmp, frictiny);

  return;
}


int num_nonzero_elem(HashTable *El_Table, int type) {
  int num = 0;
  HashEntryPtr currentPtr;
  Element *Curr_El;
  HashEntryPtr *buck = El_Table->getbucketptr();

  for (int i = 0; i < El_Table->get_no_of_buckets(); i++)
    if (*(buck + i)) {
      currentPtr = *(buck + i);
      while (currentPtr) {
        Curr_El = (Element*) (currentPtr->value);
        if (Curr_El->get_adapted_flag() == type)
          num++;
        currentPtr = currentPtr->next;
      }
    }

  return (num);
}

void reverse_states(HashTable* El_Table, vector<Jacobian*>* solHyst
    , int iter) {
  HashEntryPtr currentPtr;
  Element *Curr_El;
  HashEntryPtr *buck = El_Table->getbucketptr();

#ifdef DEBUG
  if (checkElement(El_Table))
  exit(22);
  for (int i = 0; i < nonz1; i++) {
    dbgvec[i] = 0;
    pass[i] = 0;
  }
  for (int i = 0; i < El_Table->get_no_of_buckets(); i++) {
    if (*(buck + i)) {
      currentPtr = *(buck + i);
      while (currentPtr) {
        Curr_El = (Element*) (currentPtr->value);
        currentPtr = currentPtr->next;
        if (Curr_El->get_adapted_flag() > 0) {

          int index = Curr_El->get_sol_rec_ind();
          dbgvec[index] += 1;
          pass[index] = 1;

        }
      }
    }
  }
  for (int i = 0; i < nonz1; i++) {
```

```
275       if (dbgvec[i] != 1) {
            cout << "these elements have problem:    " << i << endl
277         << "the value is  " << dbgvec[i] << endl;
            exit(EXIT_FAILURE);
279
          } else if (pass[i] != 1) {
281         cout << "this index has not been passed  " << i << endl;
            exit(EXIT_FAILURE);
283
          }
285     }

287     delete[] dbgvec;

289     cout << "number of elements after unrefinement"
        << num_nonzero_elem(El_Table) << endl;
291 //    getchar();
        if (checkElement(El_Table))
293     exit(22);
    #endif
295
        for (int i = 0; i < El_Table->get_no_of_buckets(); i++) {
297       if (*(buck + i)) {
            currentPtr = *(buck + i);
299         while (currentPtr) {
              Curr_El = (Element*) (currentPtr->value);
301           if (Curr_El->get_adapted_flag() > 0) {
                Jacobian *jacobian = solHyst->at(Curr_El->get_sol_rec_ind
      ());
303
                if (iter != 0)
305               jacobian->rev_state_vars(Curr_El, iter);

307           }
              currentPtr = currentPtr->next;
309         }
          }
311     }

313     return;
    }
315
    void print_jacobian(HashTable* El_Table, vector<Jacobian*>* solHyst
        , int iter) {
317
        HashEntryPtr currentPtr;
319     Element *Curr_El;
        HashEntryPtr *buck = El_Table->getbucketptr();
321
        for (int i = 0; i < El_Table->get_no_of_buckets(); i++) {
323       if (*(buck + i)) {
            currentPtr = *(buck + i);
325         while (currentPtr) {
              Curr_El = (Element*) (currentPtr->value);
327           if (Curr_El->get_adapted_flag() > 0) {
                Jacobian *jacobian = solHyst->at(Curr_El->get_sol_rec_ind
      ());
329             jacobian->print_jacobian(iter);
              }
331           currentPtr = currentPtr->next;
            }
333       }
```

```
  }
  return;
}

void compute_functional(HashTable* El_Table, double* functional,
    TimeProps* timeprops_ptr) {

  HashEntryPtr currentPtr;
  Element *Curr_El;
  HashEntryPtr *buck = El_Table->getbucketptr();
  double const *dx;
  double const *state_vars;
  double const *prev_state_vars;
  double dt;

  dt = timeprops_ptr->dt.at(timeprops_ptr->iter - 1);

  printf("iter=%4d  dt=%8f \n", timeprops_ptr->iter, dt);

  //we do not have make it zero here, because we want to compute
    the integration over the time and space
  //*functional = 0.0;

  for (int i = 0; i < El_Table->get_no_of_buckets(); i++)
    if (*(buck + i)) {
      currentPtr = *(buck + i);
      while (currentPtr) {
        Curr_El = (Element*) (currentPtr->value);

        if (Curr_El->get_adapted_flag() > 0) {

          dx = Curr_El->get_coord();
          state_vars = Curr_El->get_state_vars();
          prev_state_vars = Curr_El->get_prev_state_vars();

          // we used trapezoidal integration on time
          // flag is for time step 0

          *functional += .5
              * (state_vars[0] * state_vars[0]
                  + prev_state_vars[0] * prev_state_vars[0]) * dx
    [0] * dx[1]
              * dt;

        }
        currentPtr = currentPtr->next;
      }
    }

  cout << "functional is: " << *functional << endl;

  return;
}
```

C.2 calc_jacobian.C

```
#ifdef HAVE_CONFIG_H
# include <config.h>
#endif
```

```
4  #include "../header/hpfem.h"

6  #define KEY0    3797155840
   #define KEY1    0
8  #define ITER    7
   #define EFFELL  0
10 #define J       0
   #define JACIND  0
12
   //#define DEBUG
14
   void reset_resflag(ResFlag resflag[5]);
16
   void calc_jacobian(HashTable* El_Table, HashTable* NodeTable,
18     vector<Jacobian*>* solHyst, MatProps* matprops_ptr,
       TimeProps* timeprops_ptr, MapNames *mapname_ptr, double const
       increment) {
20
     int myid, numprocs;
22   MPI_Comm_rank(MPLCOMM_WORLD, &myid);
     MPI_Comm_size(MPLCOMM_WORLD, &numprocs);
24
     int neigh_flag;
26   HashEntryPtr* buck = El_Table->getbucketptr();
     HashEntryPtr currentPtr;
28   Element* Curr_El = NULL;
     Element *neigh_elem = NULL;
30
     int iter = timeprops_ptr->iter;
32   double tiny = GEOFLOW_TINY;

34   cout << "computing jacobian for time iteration " << iter << endl;

36   //here are some dummy values that we need for calc_edge_state
     int order_flag = 1;
38   double outflow = 0;

40   //this array holds ResFlag for element itself and its neighbors
     ResFlag resflag[5];
42   reset_resflag(resflag);

44   // after updating state_vars on cells we need to compute the
       fluxes
     calc_edge_states(El_Table, NodeTable, matprops_ptr, timeprops_ptr
       , myid,
46       &order_flag, &outflow, resflag[0]);

48   for (int i = 0; i < El_Table->get_no_of_buckets(); i++) {
       if (*(buck + i)) {
50       currentPtr = *(buck + i);
         while (currentPtr) {
52         Curr_El = (Element*) (currentPtr->value);

54         if (Curr_El->get_adapted_flag() > 0) {

56           int boundary = 0;

58           //this part handles if the Curr_El is a boundary element
             //cout<<"I am running do not worry"<<endl;
60           for (int neighnum = 0; neighnum < 4; neighnum++)
               if (*(Curr_El->get_neigh_proc() + neighnum) == INIT) {
62               boundary = 1;
```

```
              break;
64          }
          if (!boundary) {
66
            Jacobian *jacobian = solHyst->at(Curr_El->
    get_sol_rec_ind());
68
            double *state_vars = Curr_El->get_state_vars();
70          double *prev_state_vars = Curr_El->get_prev_state_vars
    ();
            double *gravity = Curr_El->get_gravity();
72          double *d_gravity = Curr_El->get_d_gravity();
            double *curvature = Curr_El->get_curvature();
74          Curr_El->calc_stop_crit(matprops_ptr); //this function
    updates bedfric properties
            double bedfrict = Curr_El->get_effect_bedfrict();
76
            double *dx = Curr_El->get_dx();
78          double kactxy[DIMENSION];
            double orgSrcSgn[2];
80
            Curr_El->get_slopes_prev(El_Table, NodeTable,
    matprops_ptr->gamma); // we run this to update d_state_vars
82
            if (timeprops_ptr->iter < 51)
84            matprops_ptr->frict_tiny = 0.1;
            else
86            matprops_ptr->frict_tiny = 0.000000001;

88          orgSourceSgn(Curr_El, matprops_ptr->frict_tiny,
    orgSrcSgn);

90          for (int effelement = 0; effelement < 5; effelement++)
    { //0 for the element itself, and the rest id for neighbour
    elements

92            int xp = Curr_El->get_positive_x_side(); //finding
    the direction of element
              int yp = (xp + 1) % 4, xm = (xp + 2) % 4, ym = (xp +
    3) % 4;
94            int jacmatind = jac_mat_index(effelement, xp); //this
     function returns the matrix that the jacobian matrix has to be
     stored

96            double void_res[3] = { 0., 0., 0. };

98  #ifdef DEBUG
              int gggflag = 0;
100
              if (*(Curr_El->pass_key()) == KEY0
102               && *(Curr_El->pass_key() + 1) == KEY1 && iter ==
    ITER
                  && jacmatind == JACIND)
104             gggflag = 1;
    #endif
106
              if (effelement == 0 && prev_state_vars[0] == 0.)
108
                //this is a void element so the residual vector
    does not change by changing it's values
110             jacobian->set_jacobianMat_zero(jacmatind);
```

```
112                 else if (effelement != 0
                        && void_neigh_elem(El_Table, Curr_El, effelement)
    )
114
                      //this is a void neighbor element so the residual
    of the curr_el does not depend on this neighbor
116                   jacobian->set_jacobianMat_zero(jacmatind);

118                 else {

120                   for (int j = 0; j < 4; j++) { //there is a problem
    here: I do not need to compute for first the component of
    adjoint

122                     if (j != 1) { //since we don't want to do that
    for first component of adjoint

124                       double dummydt = 0.;//this is dummy because it
    is needed in clac_edge state->zdirflux->calc_wetness_factor
    which is useless here

126                       //Attention make sure that NUM_STATE_VARS are
    selected correctly
                          //Actually we just need 3, but there is an
    excessive for first adjoint
128                       const int state_num = NUM_STATE_VARS - 2;

130                       Node* nxp = (Node*) NodeTable->lookup(
                              Curr_El->getNode() + (xp + 4) * 2);
132
                          Node* nyp = (Node*) NodeTable->lookup(
134                           Curr_El->getNode() + (yp + 4) * 2);

136                       Node* nxm = (Node*) NodeTable->lookup(
                              Curr_El->getNode() + (xm + 4) * 2);
138
                          Node* nym = (Node*) NodeTable->lookup(
140                           Curr_El->getNode() + (ym + 4) * 2);

142                       double vec_res[3];
                          double total_res[3] = { 0., 0., 0. };
144
                          int scheme = 0;
146                       for (; scheme < 2; scheme++) {

148                         //this flag shows that the pileheight before
    adding the increment is below or above the GEOFLOW_TINY.
                          //if it is below GEOFLOW_TINY then there is
    no need to update fluxes and kactxy
150                       int updateflux, srcflag;
                          reset_resflag(resflag);
152 #ifdef DEBUG
                          int dbgflag = 0, printflag = 0;
154                       double fluxxpold[state_num], fluxypold[
    state_num]; //we just need to compute jacobian for h,u,v not
    for cont. adjoint so we don't store the fluxes for the adjoint
                          double fluxxmold[state_num], fluxymold[
    state_num];
156
                          unsigned key[2];
158                       key[0] = *(Curr_El->pass_key());
                          key[1] = *(Curr_El->pass_key() + 1);
```

```
160
                        if (*(Curr_El->pass_key()) == KEY0
162                         && *(Curr_El->pass_key() + 1) == KEY1
                            && jacmatind == JACIND && iter == ITER &&
     j == J)
164
                          record_flux(El_Table, NodeTable, key,
    matprops_ptr,
166                           effelement, myid, fluxxpold, fluxypold,
     fluxxmold,
                              fluxymold);
168
#endif
170
                        // here we modify increment to one time
    compute forward and one time compute backward difference if it
    is necessary
172                     double signe = pow(-1., scheme);
                        double incr = signe * increment;
174                     increment_state(El_Table, Curr_El, incr,
    effelement, j,
                            &updateflux, &srcflag, resflag);
176
                        calc_flux_slope_kact(El_Table, NodeTable,
    Curr_El,
178                         matprops_ptr, myid, effelement,
    updateflux, srcflag,
                            resflag);
180
                        double dt = timeprops_ptr->dt.at(iter - 1);
    //at final time step we do not need the computation of adjoint
    and we always compute it for the previouse time so we need iter
    .
182                     double dtdx = dt / dx[0];
                        double dtdy = dt / dx[1];
184
                        //Attention make sure that NUM_STATE_VARS are
     selected correctly
186                     //Actually we just need 3, but there is an
    excessive for first adjoint
                        //const int state_num=NUM_STATE_VARS-2;
188                     double fluxxp[state_num], fluxyp[state_num];
    //we just need to compute jacobian for h,u,v not for cont.
    adjoint so we don't store the fluxes for the adjoint
                        double fluxxm[state_num], fluxym[state_num];
190
                        for (int ivar = 0; ivar < state_num; ivar++)
192                       fluxxp[ivar] = nxp->flux[ivar];

194                     for (int ivar = 0; ivar < state_num; ivar++)
                          fluxyp[ivar] = nyp->flux[ivar];
196
                        for (int ivar = 0; ivar < state_num; ivar++)
198                       fluxxm[ivar] = nxm->flux[ivar];

200                     for (int ivar = 0; ivar < state_num; ivar++)
                          fluxym[ivar] = nym->flux[ivar];
202
#ifdef DEBUG
204                     if (*(Curr_El->pass_key()) == KEY0
                            && *(Curr_El->pass_key() + 1) == KEY1
```

```
206                         && jacmatind == JACIND && iter == ITER &&
     j == J)
                       flux_debug(Curr_El, fluxxpold, fluxxmold,
    fluxypold,
208                            fluxymold, fluxxp, fluxxm, fluxyp,
    fluxym,
                           effelement, j, iter, dt);
210 #endif

212                    double *d_state_vars = Curr_El->
    get_d_state_vars();

214                    //here we compute the residuals
                       residual(vec_res, state_vars, prev_state_vars
    , fluxxp, //4
216                        fluxyp, fluxxm, fluxym, dtdx, dtdy, dt,
    d_state_vars, //7
                           (d_state_vars + NUM_STATE_VARS),
    curvature, //2
218                        matprops_ptr->intfrict, //1
                           bedfrict, gravity, d_gravity, Curr_El->
    get_kactxy(), //4
220                        matprops_ptr->frict_tiny, orgSrcSgn, incr
    , //3
                           matprops_ptr->epsilon, srcflag); //2
222
                       //we have to return everything back
224                    restore(El_Table, NodeTable, Curr_El,
    matprops_ptr,
                           effelement, j, myid, incr);
226
                       for (int ind = 0; ind < 3; ind++)
228                      total_res[ind] += signe * vec_res[ind];

230 //                   if (scheme == 0 && fabs(vec_res[0] / incr)
    < 5.
//                         && fabs(vec_res[1] / incr) < 5.
232 //                      && fabs(vec_res[2] / incr) < 5.)
//                       //this means that forward difference is
    enough, and we do not need to compute central difference
234                    break;

236                  }

238                  double jacincr = increment; //= scheme == 0 ?
    increment : 2. * increment;

240                  jacobian->set_jacobian(jacmatind, total_res, j,
                     // following term is necessary to consider the
    scheme that whether it is forward difference or central
    difference
242                      jacincr); //sets the propper components of
    the Jacobian for this element

244                }
               }
246          }
        }
248     } else {

250       // this for the element that are on the boundary
```

```
            Jacobian *jacobian = solHyst->at(Curr_El->
    get_sol_rec_ind());
252         jacobian->set_jacobian();

254       }
        }
256     currentPtr = currentPtr->next;
      }
258   }
  }
260 return;
}
262
int jac_mat_index(int effelement, int xp) {
264
  int yp = (xp + 1) % 4, xm = (xp + 2) % 4, ym = (xp + 3) % 4;
266 // effelement - 1 shows which neighbor we are processing
  if (effelement == 0)
268   return 0;
  else if (effelement - 1 == xp)
270   return 1;
  else if (effelement - 1 == yp)
272   return 2;
  else if (effelement - 1 == xm)
274   return 3;
  else if (effelement - 1 == ym)
276   return 4;
  else
278   exit(1);
}
280
//this function returns 1 if the neighbor element is void
282 int void_neigh_elem(HashTable* El_Table, Element* Curr_El, int
    effelement) {

284 Element* neigh_elem = (Element*) (El_Table->lookup(
      Curr_El->get_neighbors() + (effelement - 1) * KEYLENGTH));
286
  assert(neigh_elem);
288
  if (*(neigh_elem->get_prev_state_vars()) == 0.)
290   return 1;

292 return 0;
}
294
void restore(HashTable* El_Table, HashTable* NodeTable, Element*
    Curr_El,
296   MatProps* matprops_ptr, int effelement, int j, int myid, double
     increment) {

298 Element* neigh_elem;
  double *prev_state_vars = Curr_El->get_prev_state_vars();
300
  double dummydt = 0., outflow = 0.;
302 int order_flag = 1;

304 ResFlag resflag;
  resflag.callflag = 1;
306 resflag.lgft = 0;
```

```
308   int xp = Curr_El->get_positive_x_side(); //finding the direction
        of element
      int yp = (xp + 1) % 4, xm = (xp + 2) % 4, ym = (xp + 3) % 4;
310
      if (effelement == 0) { //this part of code add an increment to
        the state variables to find the Jacobian, but the problem is
        since it is called after correct, the increment shoud be added
        to the prev_state_vars
312
        prev_state_vars[j] -= increment; //changing the state varibales
         at the element itself
314     Curr_El->calc_edge_states(El_Table, NodeTable, matprops_ptr,
        myid, dummydt,
            &order_flag, &outflow, resflag, resflag); //chenge of the
        state_vars causes the all around fluxes change, this update xp
        ,yp
316
        if ((*(Curr_El->get_neigh_proc() + xm)) != INIT) { //we have to
         make sure that there exit an element in xm side
318
          Element* elem_xm = (Element*) (El_Table->lookup(
320           Curr_El->get_neighbors() + xm * KEYLENGTH));
          elem_xm->calc_edge_states(El_Table, NodeTable, matprops_ptr,
        myid,
322           dummydt, &order_flag, &outflow, resflag, resflag); //this
         update the flux on share edge with xm
        }
324     if ((*(Curr_El->get_neigh_proc() + ym)) != INIT) { //we have to
         make sure that there exit an element in ym side

326       Element* elem_ym = (Element*) (El_Table->lookup(
            Curr_El->get_neighbors() + ym * KEYLENGTH));
328       elem_ym->calc_edge_states(El_Table, NodeTable, matprops_ptr,
        myid,
            dummydt, &order_flag, &outflow, resflag, resflag); //this
         update the flux on share edge with ym
330     }

332   } else if ((*(Curr_El->get_neigh_proc() + (effelement - 1))) !=
        INIT) {

334     neigh_elem = (Element*) (El_Table->lookup(
          Curr_El->get_neighbors() + (effelement - 1) * KEYLENGTH));
336     *(neigh_elem->get_prev_state_vars() + j) -= increment;

338     if ((effelement - 1) == xp || (effelement - 1) == yp)
          Curr_El->calc_edge_states(El_Table, NodeTable, matprops_ptr,
        myid,
340           dummydt, &order_flag, &outflow, resflag, resflag); //if
        we change the state variables in xp or yp, just the flux at
        this element has to be updated
        else
342       neigh_elem->calc_edge_states(El_Table, NodeTable,
        matprops_ptr, myid,
            dummydt, &order_flag, &outflow, resflag, resflag); //
        otherwise the flux at the corresponding element has to be
        updated
344
      }
346   Curr_El->get_slopes(El_Table, NodeTable, matprops_ptr->gamma);
      return;
348 }
```

```
350 void calc_flux_slope_kact(HashTable* El_Table, HashTable* NodeTable
        ,
    Element* Curr_El, MatProps* matprops_ptr, int myid, int
    effelement,
352     int updateflux, int srcflag, ResFlag resflag[5]) {

354   double dummydt = 0., outflow = 0.;
    int order_flag = 1;
356
    ResFlag dummyresflag;
358   dummyresflag.callflag = 1;
    dummyresflag.lgft = 0;
360
    Element* neigh_elem;
362
    int xp = Curr_El->get_positive_x_side(); //finding the direction
      of element
364   int yp = (xp + 1) % 4, xm = (xp + 2) % 4, ym = (xp + 3) % 4;

366   Curr_El->get_slopes_prev(El_Table, NodeTable, matprops_ptr->gamma
      ); //we also have to update the d_state_vars for the current
      element

368   double *d_state_vars = Curr_El->get_d_state_vars();

370 //                      if (srcflag && effelement == 0) {
    //
372 //                        gmfggetcoef_(Curr_El->get_prev_state_vars()
        , d_state_vars,
    //                             (d_state_vars + NUM_STATE_VARS), dx,
374 //                             &(matprops_ptr->bedfrict[Curr_El->
        get_material()]),
    //                             &(matprops_ptr->intfrict), &kactxy[0],
        &kactxy[1],
376 //                             &tiny, &(matprops_ptr->epsilon));
    //
378 //                        Curr_El->put_kactxy(kactxy);
    //                        Curr_El->calc_stop_crit(matprops_ptr);
380 //                      }

382   if (effelement == 0 && updateflux) { //this part of code add an
      increment to the state variables to find the Jacobian, but the
      problem is since it is called after correct, the increment
      shoud be added to the prev_state_vars

384     Curr_El->calc_edge_states(El_Table, NodeTable, matprops_ptr,
      myid, dummydt,
          &order_flag, &outflow, resflag[0], dummyresflag); //change
      of the state_vars causes the all around fluxes change, this
      update xp ,yp
386     // earlier in this file, we made sure that this element is not
      a boundary element
      // so here we do not require to check the neighbor elements to
      make sure they are not located on the boundary
388
      Element* elem_xm = (Element*) (El_Table->lookup(
390       Curr_El->get_neighbors() + xm * KEYLENGTH));
      assert(elem_xm);
392   elem_xm->calc_edge_states(El_Table, NodeTable, matprops_ptr,
      myid, dummydt,
```

```
        &order_flag, &outflow, resflag[3], resflag[0]); //this
    update the flux on share edge with xm

    Element* elem_ym = (Element*) (El_Table->lookup(
        Curr_El->get_neighbors() + ym * KEYLENGTH));
    assert(elem_ym);
    elem_ym->calc_edge_states(El_Table, NodeTable, matprops_ptr,
    myid, dummydt,
        &order_flag, &outflow, resflag[4], resflag[0]); //this
    update the flux on share edge with ym

  } else if (effelement != 0 && updateflux) {

    if ((effelement - 1) == xp)

      Curr_El->calc_edge_states(El_Table, NodeTable, matprops_ptr,
    myid,
          dummydt, &order_flag, &outflow, resflag[0], resflag[1]);
    //if we change the state variables in xp or yp, just the flux
    at this element has to be updated
    else if ((effelement - 1) == yp)
      Curr_El->calc_edge_states(El_Table, NodeTable, matprops_ptr,
    myid,
          dummydt, &order_flag, &outflow, resflag[0], resflag[2]);

    else if ((effelement - 1) == xm) {

      neigh_elem = (Element*) (El_Table->lookup(
          Curr_El->get_neighbors() + (effelement - 1) * KEYLENGTH))
    ;
      assert(neigh_elem);

      neigh_elem->calc_edge_states(El_Table, NodeTable,
    matprops_ptr, myid,
          dummydt, &order_flag, &outflow, resflag[3], resflag[0]);
    //otherwise the flux at the corresponding element has to be
    updated
//otherwise the flux at the corresponding element has to be updated

    } else {

      neigh_elem = (Element*) (El_Table->lookup(
          Curr_El->get_neighbors() + (effelement - 1) * KEYLENGTH))
    ;
      assert(neigh_elem);

      neigh_elem->calc_edge_states(El_Table, NodeTable,
    matprops_ptr, myid,
          dummydt, &order_flag, &outflow, resflag[4], resflag[0]);
    //otherwise the flux at the corresponding element has to be
    updated
//otherwise the flux at the corresponding element has to be updated
    }

  }

  return;

}

void increment_state(HashTable* El_Table, Element* Curr_El, double
    increment,
```

```
    int effelement, int j, int* updateflux, int* srcflag, ResFlag
    resflag[5]) {
440
  *updateflux = 1;
442 *srcflag = 1;
  double *prev_state_vars = Curr_El->get_prev_state_vars();
444
  int xp = Curr_El->get_positive_x_side(); //finding the direction
    of element
446 int yp = (xp + 1) % 4, xm = (xp + 2) % 4, ym = (xp + 3) % 4;

448 if (effelement == 0) { //this part of code add an increment to
    the state variables to find the Jacobian, but the problem is
    since it is called after correct, the increment shoud be added
    to the prev_state_vars

450   if (j == 0 && prev_state_vars[j] < GEOFLOW_TINY) {
        *updateflux = 0;
452     *srcflag = 0;
        resflag[0].lgft = 1;
454   }

456   prev_state_vars[j] += increment; //changing the state varibales
     at the element itself

458 } else {

460   Element* neigh_elem = (Element*) (El_Table->lookup(
        Curr_El->get_neighbors() + (effelement - 1) * KEYLENGTH));
462   assert(neigh_elem);

464   if (j == 0 && *(neigh_elem->get_prev_state_vars() + j) <
    GEOFLOW_TINY) {
        *updateflux = 0;
466     resflag[jac_mat_index(effelement, xp)].lgft = 1;
      }
468
      *(neigh_elem->get_prev_state_vars() + j) += increment;
470

  }
472 return;
}
474
void reset_resflag(ResFlag resflag[5]) {
476
  for (int i = 0; i < 5; i++) {
478   resflag[i].callflag = 1;
      resflag[i].lgft = 0;
480 }

482 return;
}
```

C.3 calc_adjoint.C

```
1 #ifdef HAVE_CONFIG_H
  # include <config.h>
3 #endif
  #include "../header/hpfem.h"
```

```
5
#define KEY0 3916612844
7 #define KEY1 1321528399
#define ITER 1
9
void calc_adjoint_elem(HashTable* El_Table, vector<Jacobian*>*
    solHyst,
11     Element *Curr_El, int iter, int adjiter, int myid);

13 void calc_adjoint(HashTable* El_Table, vector<Jacobian*>* solHyst,
    int iter,
    int adjiter, int myid) {
15
  HashEntryPtr* buck = El_Table->getbucketptr();
17   HashEntryPtr currentPtr;
  Element* Curr_El = NULL;
19
  double aa, bb = .1;
21
  for (int i = 0; i < El_Table->get_no_of_buckets(); i++) {
23     if (*(buck + i)) {
      currentPtr = *(buck + i);
25       while (currentPtr) {
        Curr_El = (Element*) (currentPtr->value);
27
        if (Curr_El->get_adapted_flag() > 0) {
29           if (*(Curr_El->pass_key()) == KEY0
              && *(Curr_El->pass_key() + 1) == KEY1 && iter == ITER
    )
31             aa = bb;

33           calc_adjoint_elem(El_Table, solHyst, Curr_El, iter,
    adjiter, myid);
        }
35         currentPtr = currentPtr->next;
      }
37     }
  }
39   return;
}
41
void calc_adjoint_elem(HashTable* El_Table, vector<Jacobian*>*
    solHyst,
43     Element *Curr_El, int iter, int adjiter, int myid) {

45   double* adjoint = (Curr_El->get_state_vars() + 6);
  Jacobian *jacobian, *neighjac;
47   jacobian = solHyst->at(Curr_El->get_sol_rec_ind());

49   if (adjiter == 0) {

51     for (int i = 0; i < 3; ++i)
      adjoint[i] = *(jacobian->get_funcsens(iter) + i);
53
  } else {
55
    Element *neigh_elem;
57     double* adjoint_pointer;
    double adjcontr[3] = { 0.0, 0.0, 0.0 };
59     double*** jacobianmat;
```

```
61      for (int effelement = 0; effelement < 5; effelement++) { //0
        for the element itself, and the rest id for neighbour elements

63        if (effelement == 0) {          //this part of code

65          adjoint_pointer = (Curr_El->get_prev_state_vars() + 6);

67          jacobianmat = jacobian->get_jacobian();

69          for (int k = 0; k < 3; ++k)
              for (int l = 0; l < 3; ++l)
71              adjcontr[k] += adjoint_pointer[l] * jacobianmat[0][k][l
        ];

73        } else {

75          neigh_elem = Curr_El->get_side_neighbor(El_Table,
        effelement - 1);//basically we are checking all neighbor
        elements, and start from xp neighbor
            if (neigh_elem) {
77
              adjoint_pointer = (neigh_elem->get_prev_state_vars() + 6)
        ;
79            neighjac = solHyst->at(neigh_elem->get_sol_rec_ind());
              jacobianmat = neighjac->get_jacobian();
81
              int jacind;
83
              switch (effelement) {
85            case 1: //in xp neighbor I have to read jacobian of xm,
        because position of curr_el is in xm side of that neighbor
                jacind = 3;
87              break;
              case 2:       //for yp return ym
89              jacind = 4;
                break;
91            case 3:       //for xm return xp
                jacind = 1;
93              break;
              case 4:       //for ym return yp
95              jacind = 2;
                break;
97            default:
                cout << "invalid neighbor position" << endl;
99            }

101           for (int k = 0; k < 3; ++k)
                for (int l = 0; l < 3; ++l)
103               adjcontr[k] += adjoint_pointer[l] * jacobianmat[
        jacind][k][l];

105         }
          }
107     }

109     for (int j = 0; j < 3; j++)
          adjoint[j] = *(jacobian->get_funcsens(iter-1) + j) - adjcontr
        [j];
111   }

113   for (int i = 0; i < 3; i++)
        if (isnan(adjoint[i]) || isinf(adjoint[i]))
```

```
115       cout << "it is incorrect  " << endl;

117   return;
}
```

C.4 error_compute.C

```
#ifdef HAVE_CONFIG_H
2 # include <config.h>
#endif
4 #include "../header/hpfem.h"

6 #define KEY0   3788876458
#define KEY1   2863311530
8 #define ITER   5

10 void error_compute(HashTable* El_Table, HashTable* NodeTable,
     TimeProps* timeprops_ptr, MatProps* matprops_ptr, int iter, int
      myid,
12     int numprocs) {

14   setup_geoflow(El_Table, NodeTable, myid, numprocs, matprops_ptr,
       timeprops_ptr);
16
   int order_flag = 1; //this is dummy here
18   double outflow[1];  //this is dummy here
   ResFlag resflag;
20   resflag.callflag = 1;
   resflag.lgft = 0;
22   calc_edge_states(El_Table, NodeTable, matprops_ptr, timeprops_ptr
     , myid,
       &order_flag, outflow, resflag);
24
   HashEntryPtr* buck = El_Table->getbucketptr();
26   HashEntryPtr currentPtr;
   Element* Curr_El = NULL;
28
   if (iter != 0) {
30     for (int i = 0; i < El_Table->get_no_of_buckets(); i++)
       if (*(buck + i)) {
32         currentPtr = *(buck + i);
         while (currentPtr) {
34           Curr_El = (Element*) (currentPtr->value);
           if (Curr_El->get_adapted_flag() == NEWSON) {
36
             int dbgflag;
38             if (*(Curr_El->pass_key()) == KEY0
                 && *(Curr_El->pass_key() + 1) == KEY1 && iter ==
     ITER)
40               dbgflag = 1;

42             double *state_vars = Curr_El->get_state_vars();
             double *prev_state_vars = Curr_El->get_prev_state_vars
     ();
44             double *gravity = Curr_El->get_gravity();
             double *d_gravity = Curr_El->get_d_gravity();
46             double *curvature = Curr_El->get_curvature();
             Curr_El->calc_stop_crit(matprops_ptr); //this function
     updates bedfric properties
```

```
48              double bedfrict = Curr_El->get_effect_bedfrict();
                double velocity[DIMENSION];
50              double *dx = Curr_El->get_dx();
                double kactxy[DIMENSION];
52              double orgSrcSgn[2], vec_res[3];

54              Curr_El->get_slopes_prev(El_Table, NodeTable,
    matprops_ptr->gamma);
                double *d_state_vars = Curr_El->get_d_state_vars();
56
                if (timeprops_ptr->iter < 50)
58                matprops_ptr->frict_tiny = 0.1;
                else
60                matprops_ptr->frict_tiny = 0.000000001;

62              orgSourceSgn(Curr_El, matprops_ptr->frict_tiny,
    orgSrcSgn);

64              double dt = timeprops_ptr->dt.at(iter - 1); // if we
    have n iter size of dt vector is n-1
                double dtdx = dt / dx[0];
66              double dtdy = dt / dx[1];

68              double *el_error = Curr_El->get_el_error();

70              double *constAdj = Curr_El->get_const_adj();
                double *correction = Curr_El->get_correction();
72
                int xp = Curr_El->get_positive_x_side(); //finding the
    direction of element
74              int yp = (xp + 1) % 4, xm = (xp + 2) % 4, ym = (xp + 3)
     % 4;

76              const int state_num = NUM_STATE_VARS - 2;

78              Node* nxp = (Node*) NodeTable->lookup(
                    Curr_El->getNode() + (xp + 4) * 2);
80
                Node* nyp = (Node*) NodeTable->lookup(
82                  Curr_El->getNode() + (yp + 4) * 2);

84              Node* nxm = (Node*) NodeTable->lookup(
                    Curr_El->getNode() + (xm + 4) * 2);
86
                Node* nym = (Node*) NodeTable->lookup(
88                  Curr_El->getNode() + (ym + 4) * 2);

90              double fluxxp[state_num], fluxyp[state_num]; //we just
    need to compute jacobian for h,u,v not for cont. adjoint so we
    don't store the fluxes for the adjoint
                double fluxxm[state_num], fluxym[state_num];
92
                for (int ivar = 0; ivar < state_num; ivar++)
94                fluxxp[ivar] = nxp->flux[ivar];

96              for (int ivar = 0; ivar < state_num; ivar++)
                  fluxyp[ivar] = nyp->flux[ivar];
98
                for (int ivar = 0; ivar < state_num; ivar++)
100               fluxxm[ivar] = nxm->flux[ivar];

102             for (int ivar = 0; ivar < state_num; ivar++)
```

```
                fluxym[ivar] = nym->flux[ivar];

            if (*(Curr_El->pass_key()) == KEY0
                && *(Curr_El->pass_key() + 1) == KEY1 && iter ==
    ITER)
              dbgflag = 1;

            residual(vec_res, state_vars, prev_state_vars, fluxxp,
    fluxyp,
                fluxxm, fluxym, dtdx, dtdy, dt, d_state_vars,
                (d_state_vars + NUM_STATE_VARS), curvature,
                matprops_ptr->intfrict, bedfrict, gravity,
    d_gravity,
                Curr_El->get_kactxy(), matprops_ptr->frict_tiny,
    orgSrcSgn,
                0./*here increment is zero*/, matprops_ptr->epsilon
    );

            state_vars[1] = vec_res[0];
            state_vars[4] = vec_res[1];
            state_vars[5] = vec_res[2];

            el_error[1] = 0.0;
            *correction = 0.0;
            for (int j = 0; j < 3; j++) {
              el_error[1] += vec_res[j]
                  * (state_vars[NUM_STATE_VARS + j] - constAdj[j]);

              *correction += vec_res[j] * state_vars[NUM_STATE_VARS
     + j];
            }
            //if (el_error[1]!=0)
            //  cout<<"it should print blue"<<endl ;

          }
          currentPtr = currentPtr->next;
        }
      }
  }

#ifdef DEBUG
  if (checkElement(El_Table))
  exit(22);
//    (timeprops_ptr->iter)++;
//    tecplotter(El_Table, NodeTable, matprops_ptr, timeprops_ptr,
    mapname_ptr,
//        dummyv_star, adjflag);
  cout << "number of elements -7   " << num_nonzero_elem(El_Table,
    -7) << endl
  << "number of elements -6   " << num_nonzero_elem(El_Table, -6)
    << endl
  << "number of elements  0   " << num_nonzero_elem(El_Table, 0) <<
    endl
  << "number of elements  1   " << num_nonzero_elem(El_Table, 1) <<
    endl
  << "number of elements  2   " << num_nonzero_elem(El_Table, 2) <<
    endl
  << "number of elements  3   " << num_nonzero_elem(El_Table, 3) <<
    endl
  << "number of elements  4   " << num_nonzero_elem(El_Table, 4) <<
    endl
```

```
150   << "number of elements  5   " << num_nonzero_elem(El_Table, 5) <<
      endl;
//     getchar();
152   int nonz = num_nonzero_elem(El_Table);

154   cout << "number of elements after refinement  " << nonz << endl;

156   for (int i = 0; i < El_Table->get_no_of_buckets(); i++) {
        if (*(buck + i)) {
158       currentPtr = *(buck + i);
          while (currentPtr) {
160         Curr_El = (Element*) (currentPtr->value);
            currentPtr = currentPtr->next;
162         if (Curr_El->get_adapted_flag() > 0) {

164           int index = Curr_El->get_sol_rec_ind();
              dbgvec[index] += 1;
166           pass[index] = 1;

168         }
          }
170     }
      }
172
      for (int i = 0; i < nonz1; i++) {
174     if (dbgvec[i] != 4) {
          cout << "these elements have problem:   " << i << endl
176       << "the value is  " << dbgvec[i] << endl;
          exit(EXIT_FAILURE);
178
        } else if (pass[i] != 1) {
180       cout << "this index has not been passed  " << i << endl;
          exit(EXIT_FAILURE);
182
        }
184   }
#endif
186   return;
}
```

C.5 residual.C

```
1  #ifdef HAVE_CONFIG_H
   # include <config.h>
3  #endif
   #include "../header/hpfem.h"
5
   #define DEBUG
7
   void residual(double* residual, double *state_vars, double *
       prev_state_vars, //3
9      double *fluxxp, double *fluxyp, double *fluxxm, double *fluxym,
        double dtdx, //5
       double dtdy, double dt, double *d_state_vars_x, double *
       d_state_vars_y, //4
11     double *curvature, double intfrictang, double bedfrict, double
       *gravity, //4
       double *dgdx, double* kactxyelem, double fric_tiny, double*
       orgSrcSgn, //4
```

```
13      double increment, double epsilon, int srcflag) {

15    double velocity[DIMENSION];
      double kactxy[DIMENSION];
17    //double bedfrict;

19    if (prev_state_vars[0] > GEOFLOW_TINY) {
        for (int k = 0; k < DIMENSION; k++)
21        kactxy[k] = kactxyelem[k];

23      if ((prev_state_vars[2] == 0. && prev_state_vars[3] ==
      increment)
            || (prev_state_vars[3] == 0. && prev_state_vars[2] ==
      increment)) {
25        velocity[0] = 0.;
          velocity[1] = 0.;
27      } else {

29        // fluid velocities
          velocity[0] = prev_state_vars[2] / prev_state_vars[0];
31        velocity[1] = prev_state_vars[3] / prev_state_vars[0];
        }
33
      } else {
35      for (int k = 0; k < DIMENSION; k++) {
          kactxy[k] = epsilon;
37        velocity[k] = 0.;
        }
39      //bedfrict = bedfrictin;
      }
41
      for (int i = 0; i < 3; i++)
43      residual[i] = 0.0;

45    residual[0] = state_vars[0] - prev_state_vars[0]
          + dtdx * (fluxxp[0] - fluxxm[0]) + dtdy * (fluxyp[0] - fluxym
        [0]);
47    residual[1] = state_vars[2] - prev_state_vars[2]
          + dtdx * (fluxxp[2] - fluxxm[2]) + dtdy * (fluxyp[2] - fluxym
        [2]);
49    residual[2] = state_vars[3] - prev_state_vars[3]
          + dtdx * (fluxxp[3] - fluxxm[3]) + dtdy * (fluxyp[3] - fluxym
        [3]);
51
      if (prev_state_vars[0] > GEOFLOW_TINY && srcflag) {
53
        double unitvx = 0., unitvy = 0., h_inv = 0., speed = 0.;
55
        speed = sqrt(velocity[0] * velocity[0] + velocity[1] * velocity
        [1]);
57
        if (speed > 0.) {
59        unitvx = velocity[0] / speed;
          unitvy = velocity[1] / speed;
61      }

63      //x dir
        double s1 = gravity[0] * prev_state_vars[0];
65
        double sin_int_fric = sin(intfrictang);
67      double s2 = orgSrcSgn[0] * prev_state_vars[0] * kactxy[0]
```

```
        * (gravity[2] * d_state_vars_y[0] + dgdx[1] *
    prev_state_vars[0])
        * sin_int_fric;
    double tan_bed_fric = tan(bedfrict);
    double s3 = unitvx
        * max(
            gravity[2] * prev_state_vars[0]
                + velocity[0] * prev_state_vars[2] * curvature[0],
    0.0)
        * tan_bed_fric;

    residual[1] -= dt * (s1 - s2 - s3);

    //y dir

    s1 = gravity[1] * prev_state_vars[0];

    s2 = orgSrcSgn[1] * prev_state_vars[0] * kactxy[0]
        * (gravity[2] * d_state_vars_x[0] + dgdx[0] *
    prev_state_vars[0])
        * sin_int_fric;

    s3 = unitvy
        * max(
            gravity[2] * prev_state_vars[0]
                + velocity[1] * prev_state_vars[3] * curvature[1],
    0.0)
        * tan_bed_fric;

    residual[2] -= dt * (s1 - s2 - s3);
  }

#ifdef DEBUG

//  for (int k = 0; k < 3; k++)
//    if (residual[k] > 1e-5) {
//      cout << "something that has to be checked" << endl << flush
    ;
//      exit(-2);
//    }

  for (int k = 0; k < 3; k++)
    if (isnan(residual[k])) {
      cout << "exit for NAN in residual" << endl << flush;
      exit(-1);
    }

  for (int k = 0; k < 3; k++)
    if (isinf(residual[k])) {
      cout << "exit for Inf in residual" << endl << flush;
      exit(-2);
    }
#endif

  return;
}
```

C.6 uniform_refine.C

```
#ifdef HAVE_CONFIG_H
# include <config.h>
#endif
#include "../header/hpfem.h"

#define KEY0   3788876458
#define KEY1   2863311530
#define ITER   5

void uinform_refine(HashTable* El_Table, HashTable* NodeTable,
    TimeProps* timeprops_ptr, MatProps* matprops_ptr, int numprocs,
     int myid) {

  HashEntryPtr* buck = El_Table->getbucketptr();
  HashEntryPtr currentPtr;
  Element* Curr_El = NULL;
  int rescomp = 1;

  //for debugging perpose
  unsigned key[2] = { KEY0, KEY1 };
  double max=0;

#ifdef DEBUG
  double dummyv_star = 0.0;
  int adjflag = 1;
  tecplotter(El_Table, NodeTable, matprops_ptr, timeprops_ptr,
    mapname_ptr,
      dummyv_star, adjflag);
  int nonz1 = num_nonzero_elem(El_Table);

  cout << "number of elements before refinement  " << nonz1 << endl
    ;

  int *dbgvec = new int[nonz1];
  int *pass = new int[nonz1];
  for (int i = 0; i < nonz1; i++) {
    dbgvec[i] = 0;
    pass[i] = 0;
  }
  if (checkElement(El_Table))
  exit(23);
#endif

//  if (checkElement(El_Table, &max, key))
//    cout << "here is the problem" << endl;

  htflush(El_Table, NodeTable, 1);
  move_data(numprocs, myid, El_Table, NodeTable, timeprops_ptr);

  for (int i = 0; i < El_Table->get_no_of_buckets(); i++) {
    if (*(buck + i)) {
      currentPtr = *(buck + i);
      while (currentPtr) {
        Curr_El = (Element*) (currentPtr->value);
        if (Curr_El->get_adapted_flag() >= NOTRECADAPTED) {
          Curr_El->put_adapted_flag(NOTRECADAPTED);
        }
        currentPtr = currentPtr->next;
      }
    }
  }

```

```
  ElemPtrList RefinedList(num_nonzero_elem(El_Table));

  for (int i = 0; i < El_Table->get_no_of_buckets(); i++) {
    if (*(buck + i)) {
      currentPtr = *(buck + i);
      while (currentPtr) {
        Curr_El = (Element*) (currentPtr->value);
        currentPtr = currentPtr->next;
        if (Curr_El->get_adapted_flag() == NOTRECADAPTED) {

          refinewrapper(El_Table, NodeTable, matprops_ptr, &
    RefinedList,
              Curr_El, rescomp);
        }

      }
    }
  }
//  if (checkElement(El_Table,&max, key))
//    cout << "here is the problem" << endl;

#ifdef DEBUG
  if (checkElement(El_Table))
  exit(24);
  cout << "number of elements -7   " << num_nonzero_elem(El_Table,
    -7) << endl
  << "number of elements -6   " << num_nonzero_elem(El_Table, -6)
    << endl
  << "number of elements  0   " << num_nonzero_elem(El_Table, 0) <<
     endl
  << "number of elements  1   " << num_nonzero_elem(El_Table, 1) <<
     endl
  << "number of elements  2   " << num_nonzero_elem(El_Table, 2) <<
     endl
  << "number of elements  3   " << num_nonzero_elem(El_Table, 3) <<
     endl
  << "number of elements  4   " << num_nonzero_elem(El_Table, 4) <<
     endl
  << "number of elements  5   " << num_nonzero_elem(El_Table, 5) <<
     endl;
#endif

  bilinear_interp(El_Table);  //this function reconstruct linear
    interpolation

//  if (checkElement(El_Table, &max, key))
//    cout << "here is the problem" << endl;

  refine_neigh_update(El_Table, NodeTable, numprocs, myid, (void*)
    &RefinedList,
      timeprops_ptr); //this function delete old father elements

//  if (checkElement(El_Table, &max, key))
//    cout << "here is the problem" << endl;
  RefinedList.trashlist();

  move_data(numprocs, myid, El_Table, NodeTable, timeprops_ptr);

//  if (checkElement(El_Table, &max, key))
//    cout << "here is the problem" << endl;
  return;
}
```

C.7 bilinear_interp.C

```
#ifdef HAVE_CONFIG_H
# include <config.h>
#endif
#include "../header/hpfem.h"

void bilinear_interp(HashTable* El_Table) {

  HashEntryPtr currentPtr;
  Element *Curr_El, *father, *elem11, *elem12, *elem21, *elem22;
  HashEntryPtr *buck = El_Table->getbucketptr();
  int which_son;

  for (int i = 0; i < El_Table->get_no_of_buckets(); i++)
    if (*(buck + i)) {
      currentPtr = *(buck + i);
      while (currentPtr) {
        Curr_El = (Element*) (currentPtr->value);
        if (Curr_El->get_adapted_flag() == NEWSON) {
          father = (Element*) El_Table->lookup(Curr_El->getfather()
    );
          assert(father->get_adapted_flag()==OLDFATHER);

          which_son = Curr_El->get_which_son();
#ifdef DEBUG
          double aa, bb = .1;
          if (*(Curr_El->pass_key()) == KEY0
              && *(Curr_El->pass_key() + 1) == KEY1)
          aa = bb;
#endif
          switch (which_son) {

          case 0: {
            elem22 = father;
            elem12 = father->get_side_neighbor(El_Table, 2);
            elem21 = father->get_side_neighbor(El_Table, 3);
            elem11 = father->get_side_neighbor(El_Table, 6);
            bilinear_interp_elem(elem11, elem21, elem12, elem22,
    Curr_El);
          }
            break;
          case 1: {
            elem22 = father->get_side_neighbor(El_Table, 0);
            elem12 = father;
            elem21 = father->get_side_neighbor(El_Table, 7);
            elem11 = father->get_side_neighbor(El_Table, 3);
            bilinear_interp_elem(elem11, elem21, elem12, elem22,
    Curr_El);

          }
            break;
          case 2: {
            elem22 = father->get_side_neighbor(El_Table, 4);
            elem12 = father->get_side_neighbor(El_Table, 1);
            elem21 = father->get_side_neighbor(El_Table, 0);
            elem11 = father;
```

```
            bilinear_interp_elem(elem11, elem21, elem12, elem22,
    Curr_El);

          }
            break;
          case 3: {
            elem22 = father->get_side_neighbor(El_Table, 1);
            elem12 = father->get_side_neighbor(El_Table, 5);
            elem21 = father;
            elem11 = father->get_side_neighbor(El_Table, 2);
            bilinear_interp_elem(elem11, elem21, elem12, elem22,
    Curr_El);

          }
            break;
          default:
            cout << "incorrect son please check me" << endl;

          }

        }
        currentPtr = currentPtr->next;
      }
    }
  return;
}

double bilinear_interp_value(double x1, double x2, double y1,
    double y2,
    double f11, double f21, double f12, double f22, double xinterp,
    double yinterp, int type) {

//  xmin = x1
//  xmax = x2
//  ymin = y1
//  ymax = y2
//
//  f12--------f22
//  |          |
//  |          |
//  |  interp  |
//  |          |
//  |          |
//  f11--------f21

  double interp = 0.0;
  switch (type) {
  case 0:     //for bilinear interpolation
    interp = (f11 * (x2 - xinterp) * (y2 - yinterp)
        + f21 * (xinterp - x1) * (y2 - yinterp)
        + f12 * (x2 - xinterp) * (yinterp - y1)
        + f22 * (xinterp - x1) * (yinterp - y1)) / ((x2 - x1) * (y2
    - y1));
    break;
  case 1:
    //interpolation in y
    //no other modification on bilinear interpolation is required,
    //since we call the function such that put zero for those
    elements that do not exist
    //we just removed (x2 - x1) from denominator
    interp = (f11 * (x2 - xinterp) * (y2 - yinterp)
        + f21 * (xinterp - x1) * (y2 - yinterp)
```

```
110         + f12 * (x2 - xinterp) * (yinterp - y1)
            + f22 * (xinterp - x1) * (yinterp - y1)) / (y2 - y1);
112       break;
    case 2:
114       //interpolation in x
          //no other modification on bilinear interpolation is required,
116       //since we call the function such that put zero for those
          elements that do not exist
          //we just removed (y2 - y1) from denominator
118       interp = (f11 * (x2 - xinterp) * (y2 - yinterp)
            + f21 * (xinterp - x1) * (y2 - yinterp)
120         + f12 * (x2 - xinterp) * (yinterp - y1)
            + f22 * (xinterp - x1) * (yinterp - y1)) / (x2 - x1);
122       break;

124   default:
        cout << "not a valid type in interp_value function " << endl;
126   }
    if (isnan(interp) || isinf(interp))
128   cout << "it is so sad that I found you" << endl;
    return (interp);
130 }

132 void bilinear_interp_elem(Element *elem11, Element *elem21, Element
        *elem12,
      Element *elem22, Element *Curr_El) {
134
    double *state_vars, *elem11_state, *elem12_state, *elem21_state,
136     *elem22_state;
    double *prev_state_vars, *elem11_prev_state, *elem12_prev_state,
138     *elem21_prev_state, *elem22_prev_state;
    double *coord, *elem11_coord, *elem12_coord, *elem21_coord, *
      elem22_coord;
140
    int type = 0;     //this is just a flag that indicates the type
      of element
142
    state_vars = Curr_El->get_state_vars();
144 prev_state_vars = Curr_El->get_prev_state_vars();
    coord = Curr_El->get_coord();
146
    elem11_state = elem11->get_state_vars();
148 elem12_state = elem12->get_state_vars();
    elem21_state = elem21->get_state_vars();
150 elem22_state = elem22->get_state_vars();

152 elem11_prev_state = elem11->get_prev_state_vars();
    elem12_prev_state = elem12->get_prev_state_vars();
154 elem21_prev_state = elem21->get_prev_state_vars();
    elem22_prev_state = elem22->get_prev_state_vars();
156
    elem11_coord = elem11->get_coord();
158 elem12_coord = elem12->get_coord();
    elem21_coord = elem21->get_coord();
160 elem22_coord = elem22->get_coord();

162 if (elem11 && elem12 && elem21 && elem22) {
      //this is an ordinary case for an element inside the domain
164   //type = 0; we initialized type=0
      double aaa, bbb = .1;
166   for (int j = 0; j < NUM_STATE_VARS + 3; j++)
        if ( isnan(
```

```
168             elem11_state[j]) || isnan(elem12_state[j]) || isnan(
    elem21_state[j]) || isnan(elem22_state[j]) ||
                isinf(elem11_state[j]) || isinf(elem12_state[j]) || isinf
    (elem21_state[j]) || isinf(elem22_state[j]))
170             aaa = bbb;

172         for (int j = 0; j < NUM_STATE_VARS + 3; j++) {
              state_vars[j] = bilinear_interp_value(elem11_coord[0],
    elem21_coord[0],
174             elem21_coord[1], elem22_coord[1], elem11_state[j],
    elem21_state[j],
                elem12_state[j], elem22_state[j], coord[0], coord[1],
    type);
176
              prev_state_vars[j] = bilinear_interp_value(elem11_coord[0],
178             elem21_coord[0], elem21_coord[1], elem22_coord[1],
                elem11_prev_state[j], elem21_prev_state[j],
    elem12_prev_state[j],
180             elem22_prev_state[j], coord[0], coord[1], type);
            }
182     } else if ((!elem11 && !elem12 && elem21 && elem22) //
              interpolation only in y
        || (elem11 && elem12 && !elem21 && !elem22) //left or right side
              of father is boundary
184           ) {
            type = 1;
186         if (elem11) { // in this case elem21 & elem22 do not exist, so
            we replace their value with zero
              for (int j = 0; j < NUM_STATE_VARS + 3; j++) {
188             state_vars[j] = bilinear_interp_value(0,
                    0,  //interpolation is in y, so x position is not
    important
190                 elem11_coord[1], elem12_coord[1], elem11_state[j], 0,
                    elem12_state[j], 0, coord[0], coord[1], type);
192
                prev_state_vars[j] = bilinear_interp_value(0,
194                 0,  //interpolation is in y, so x position is not
    important
                    elem11_coord[1], elem12_coord[1], elem11_prev_state[j],
     0,
196                 elem12_prev_state[j], 0, coord[0], coord[1], type);
              }
198
            } else {// in this case elem11 & elem12 do not exist, so we
            replace their value with zero
200
              for (int j = 0; j < NUM_STATE_VARS + 3; j++) {
202             state_vars[j] = bilinear_interp_value(0,
                    0,  //interpolation is in y, so x position is not
    important
204                 elem21_coord[1], elem22_coord[1], 0, elem21_state[j],
    0,
                    elem22_state[j], coord[0], coord[1], type);
206
                prev_state_vars[j] = bilinear_interp_value(0,
208                 0,  //interpolation is in y, so x position is not
    important
                    elem21_coord[1], elem22_coord[1], 0, elem21_prev_state[
    j], 0,
210                 elem22_prev_state[j], coord[0], coord[1], type);
              }
212         }
```

```
  } else if ((!elem11 && elem12 && !elem21 && elem22) //
    interpolation only in x
214 || (elem11 && !elem12 && elem21 && !elem22) //top or bottom side
    of father is boundary
      ) {
216   type = 2;

218   if (elem11) { // in this case elem12 & elem22 do not exist, so
    we replace their value with zero
        for (int j = 0; j < NUM_STATE_VARS + 3; j++) {
220       state_vars[j] = bilinear_interp_value(elem11_coord[0],
    elem21_coord[0],
            0, 0, //interpolation is in x, so y position is not
    important
222             elem11_state[j], elem21_state[j], 0, 0, coord[0], coord
    [1], type);

224         prev_state_vars[j] = bilinear_interp_value(elem11_coord[0],
              elem21_coord[0], 0,
226             0,  //interpolation is in x, so y position is not
    important
              elem11_prev_state[j], elem21_prev_state[j], 0, 0, coord
    [0],
228             coord[1], type);
        }
230
    } else {// in this case elem11 & elem21 do not exist, so we
    replace their value with zero
232
        for (int j = 0; j < NUM_STATE_VARS + 3; j++) {
234       state_vars[j] = bilinear_interp_value(elem12_coord[0],
    elem22_coord[0],
            0, 0, //interpolation is in x, so y position is not
    important
236             0, 0, elem12_state[j], elem22_state[j], coord[0], coord
    [1], type);

238         prev_state_vars[j] = bilinear_interp_value(elem12_coord[0],
              elem22_coord[0], 0,
240             0,  //interpolation is in x, so y position is not
    important
              0, 0, elem12_prev_state[j], elem22_prev_state[j], coord
    [0],
242             coord[1], type);
        }
244   }

246 } else if ((elem11 && !elem12 && !elem21 && !elem22)//father is
    in corner so there is no element for interp
  || (!elem11 && elem12 && !elem21 && !elem22)//we do not do any
    extrapolation and leave as it is,
248     || (!elem11 && !elem12 && elem21 && !elem22)//which in
    refinement constructor should be the value of father element
      || (!elem11 && !elem12 && !elem21 && elem22)) {
250   //do not do anything

252 } else {
    cout << "something is wrong in this configuration" << endl;
254 *(Curr_El->get_state_vars()) = 50;
    return;
256 }
```

```
258     return;

260 }
```

References

[1] Marian Nemec, MJ Aftosmis, and Mathias Wintzer. Adjoint-based adaptive mesh refinement for complex geometries. *AIAA Paper*, pages 1–23, 2008.

[2] S.B. Savage and K. Hutter. The motion of a finite mass of granular material down a rough incline. *Journal of Fluid Mechanics*, 199:177–215, 1989.