Appendix A

Implementation of MG parts

A.1 Jacobian and Gauss-Seidel RB

The main iterative ODE solver, in this version of the multigrid program, is a Gauss-Seidel Red-Black, in addition a Jacobian solver was developed as a stepping stone and testing purposes. It is a modification of the Jacobian method, where the updated values are used where available, which lead to a doubled converging rate **NumReci**

Our problem is given by $\nabla^2\phi = -\rho$, one way to think of the jacobian method is as a diffusion problem, and with the equilibrium solution as our wanted solution. If we then discretize the diffusion problem by a Forward-Time-Centralized-Space scheme, we arrive at the Jacobian method, which is shown explicitly below for 1 dimension.

$$\frac{\partial\phi}{\partial t} = \nabla^2\phi + \rho \tag{A.1}$$

The subscript j indicates the spatial coordinate, and the superscript n is the 'temporal' component.

$$\frac{\phi_j^{n+1} - \phi_j^{n+1}}{\Delta t} = \frac{\phi_{j+1}^n - 2\phi_j^n + \phi_{j-1}^n}{\Delta x^2} + \rho_j \tag{A.2}$$

This is numerically stable if $\Delta t/\Delta x^2 \leq 1/2$, so using the timestep $\Delta t = \Delta x^2/2$ we get

$$\phi_j^{n+1} = \phi_j^n + \frac{1}{2}\left(\phi_{j+1}^n - 2\phi_j^n + \phi_{j-1}^n\right) + \frac{\Delta x^2}{2}\rho_j \tag{A.3}$$

Then we arrive at the Jacobian method

$$\phi_j^{n+1} = \frac{1}{2}\left(\phi_{j+1}^n + \phi_{j-1}^n + \Delta x^2\rho_j\right) \tag{A.4}$$

The Gauss-Seidel method uses updated values, where available, and is given by

$$\phi_j^{n+1} = \frac{1}{2}\left(\phi_{j+1}^n + \phi_{j-1}^{n+1} + \Delta x^2 \rho_j\right) \tag{A.5}$$

Following the same procedure we get the Gauss-Seidel method for for 2 and 3 dimensions.

$$\phi_{j,k}^{n+1} = \frac{1}{4}\left(\phi_{j+1,k}^n + \phi_{j-1,k}^{n+1} + \phi_{j,k+1}^n + \phi_{j,k-1}^{n+1} + \Delta x^2 \rho_{j,k}\right) \tag{A.6}$$

$$\phi_{j,k,l}^{n+1} = \frac{1}{8}\left(\phi_{j+1,k,l}^n + \phi_{j-1,k,l}^{n+1} + \phi_{j,k+1,l}^n + \phi_{j,k-1,l}^{n+1} + \phi_{j,k,l+1}^n + \phi_{j,k,l-1}^{n+1} + \Delta x^2 \rho_{j,k,l}\right) \tag{A.7}$$

Here we have implemented a different version of the Gauss-Seidel algorithm called Red and Black ordering, which has conseptual similarities to the leapfrog algorithm, where usually position and velocity is computed at t and $t + (\delta t)/2$. Every other grid point is labeled a red point, and the remaining is black. When updating a red node only black nodes are used, and when updating black nodes only red nodes are used. Then a whole cycle consists of two halfsteps which calculates the red and black nodes seperately.

- For all red points:

$$\phi_{j,k,l}^{n+1/2} = \frac{1}{8}\left(\phi_{j+1,k,l}^n + \phi_{j-1,k,l}^n + \phi_{j,k+1,l}^n + \phi_{j,k-1,l}^n + \phi_{j,k,l+1}^n + \phi_{j,k,l-1}^n + \Delta x^2 \rho_{j,k,l}\right)$$

- For all black points:

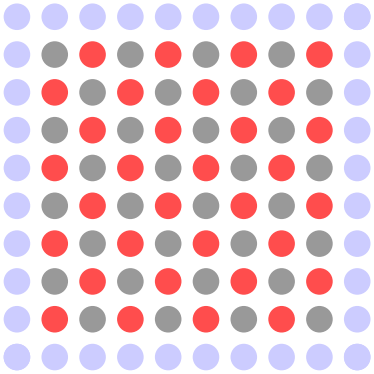
$$\phi_{j,k,l}^{n+1} = \frac{1}{8}\left(\phi_{j+1,k,l}^{n+1/2} + \phi_{j-1,k,l}^{n+1/2} + \phi_{j,k+1,l}^{n+1/2} + \phi_{j,k-1,l}^{n+1/2} + \phi_{j,k,l+1}^{n+1/2} + \phi_{j,k,l-1}^{n+1/2} + \Delta x^2 \rho_{j,k,l}\right)$$

A.1.1 Implementation

Jacobi's method

The implementation of the jacobian algorithm is straightforward, but it has the downside of slow convergence and bad smoothing properties, in addition to needing an additional created. When ϕ_i^{n+1} is computed we need access to the previous value ϕ_{i-1}^n, and more values in higher dimensions, so either the previous values need to stored seperately, or ϕ_i^{n+1} can be computed on a new grid and then copied over after completing the cycle. In this implementation we computed on a temporary grid and then copied over, since it was mostly for debugging purposes and efficiency was not a concern.

The computation is done by starting at index $g = 0$, computing the surrounding grid indexes, $gj, gjj, ...$, where gj is the next grid point along the x-axis, and



(a) Red and Black ordering

gjj is the previous value. Then the entire grid is looped trough over, incrementing both g and the surrounding grid indexes $gj, gjj, ...$ The computation on the ghost layers will be incorrect but those will be overwritten when swapping halos. See A.1.2 for an example code in 2D.

Gauss-Seidel Red and Black

In the implementation of Gauss-Seidel algorithm we use a clever ordering of the computations, called Red and Black ordering, both to increase the smoothing properties of the algorithm as well as avoiding creating a temporary grid to store ϕ^{n+1} in. Every grid point where the indexes sum up to an even number is labeled a red point and the odd index groupings are labeled black points, see fig. A.1a. Then each red point is directly surrounded by only black points and vica versa.

A cycle is then divided into 2 halfcycles, where each halfcycle computes ϕ^{n+1} for the red and black points respectively.

1. for(int c = 0; c¡nCycles; c++)
    - Cycle through red points and compute ϕ^{n+1}
    - Swap Halo
    - Cycle through black points and compute ϕ^{n+1}
    - Swap Halo

For the 2 dimensional case the cycling is done first for the odd rows and even rows seperately, due to the similarity between all the red points in the odd rows, and between the red points in the even rows. Then the cycling could be generalized into a static inline function used for all the cycling. See A.1.3 for the 2D implementation.

For the 3 dimensional case there is now did two different takes on the problem, one where the iteration through the grid is streamlined, but needing several loops

through the grid taking care of a subgroup of the grid points each loop. The other algorithm uses one loop through the grid, with different conditions on the edges to make it go through the correct grid points in each line.

When the loops are streamlined the edges the loop go through the entire grid, but when it reaches and edge in it either needs to add or subtract 1 to the iterator index. In we want to do a red pass, computing all the red values, the grid is cycled through increasing by 2 each time. Then it will access up the indexes, $36, 38, 40, 42, \cdots$. In the second row we want it to use the index 43, instead of 42, so we need to increase it by 1 when it reaches the edge. When it reaches the end of the second line, we want it to increase from 47 to 48, so then we need to subtract 1. In the next layer we need to shift the behaviour on the edges to the opposite. See

.	.	.	.	.	.
48	49	49	50	51	52
42	43	44	45	46	47
36	37	38	39	40	41

In the other implementation I tried something similar to the 2D implementation, where it does several loops through the grid, computing an easier subgroup of the red nodes each time, so the iteratior index can increase by just to each time. So for the red points it computes the odd and even layers and rows seperately.

- Compute Odd layers, odd rows
- Compute Odd layers, even rows
- Compute Even Layers, odd rows
- Compute Even layers, even rows

A.1.2 Jacobian code

```
for(int c = 0; c < nCycles; c++){
  // Index of neighboring nodes
  int gj = sizeProd[1];
  int gjj= -sizeProd[1];
  int gk = sizeProd[2];
  int gkk= -sizeProd[2];

  for(long int g = 0; g < sizeProd[rank]; g++){
    tempVal[g] = 0.25*( phiVal[gj] + phiVal[gjj] +
              phiVal[gk] + phiVal[gkk] + rhoVal[g]);

    gj++;
    gjj++;
    gk++;
    gkk++;
  }

  for(int q = 0; q < sizeProd[rank]; q++) phiVal[q] =
     tempVal[q];
  for(int d = 1; d < rank; d++) gSwapHalo(phi, mpiInfo,
     d);
}
```

Listing A.1: Code snippet 2D jacobian

A.1.3 GS-RB 2D

```
    for(int c = 0; c < nCycles;c++){

      //Increments
      int kEdgeInc = nGhostLayers[2] + nGhostLayers[rank +
          2] + sizeProd[2];

      /*************************
       *  Red Pass
       *************************/
      //Odd numbered rows
      g = nGhostLayers[1] + sizeProd[2];
      loopRedBlack2D(rhoVal, phiVal, sizeProd, trueSize,
          kEdgeInc, g, gj, gjj, gk, gkk);

      //Even numbered columns
      g = nGhostLayers[1] + 1 + 2*sizeProd[2];
      loopRedBlack2D(rhoVal, phiVal, sizeProd, trueSize,
          kEdgeInc, g, gj, gjj, gk, gkk);

      for(int d = 1; d < rank; d++) gSwapHalo(phi, mpiInfo,
          d);

      /*********************************
       *  Black pass
       *********************************/
      //Odd numbered rows
      g = nGhostLayers[1] + 1 + sizeProd[2];
      loopRedBlack2D(rhoVal, phiVal, sizeProd, trueSize,
          kEdgeInc, g, gj, gjj, gk, gkk);

      //Even numbered columns
      g = nGhostLayers[1] + 2*sizeProd[2];
      loopRedBlack2D(rhoVal, phiVal, sizeProd, trueSize,
          kEdgeInc, g, gj, gjj, gk, gkk);


      for(int d = 1; d < rank; d++) gSwapHalo(phi, mpiInfo,
          d);
    }

    return;
  }
```

Listing A.2: Main loop

```
  gj = g + sizeProd[1];
  gjj= g - sizeProd[1];
```

```
gk = g + sizeProd[2];
gkk= g - sizeProd[2];

for(int k = 1; k < trueSize[2]; k +=2){
  for(int j = 1; j < trueSize[1]; j += 2){
    phiVal[g] = 0.25*(  phiVal[gj] + phiVal[gjj] +
                phiVal[gk] + phiVal[gkk] + rhoVal[g]);
    g +=2;
    gj  +=2;
    gjj +=2;
    gk  +=2;
    gkk +=2;
  }
  g +=kEdgeInc;
  gj  +=kEdgeInc;
  gjj +=kEdgeInc;
  gk  +=kEdgeInc;
  gkk +=kEdgeInc;
}
```

Listing A.3: Loop through grid

A.2 GS-RB 3D if tests

```
/*********************
 *  Red Pass
 *********************/
g = sizeProd[3]*nGhostLayers[3];
for(int l = 0; l < trueSize[3];l++){
  for(int k = 0; k < size[2]; k++){
    for(int j = 0; j < size[1]; j+=2){
      phiVal[g] = 0.125*( phiVal[g+gj] + phiVal[g-gj] +
                phiVal[g+gk] + phiVal[g-gk] +
                phiVal[g+gl] + phiVal[g-gl] + rhoVal[g]);
      g +=2;
    }
    if(l%2){
      if(k%2) g+=1; else g-=1;
    } else {
      if(k%2) g-=1; else g+=1;
    }

  }
  if(l%2) g-=1; else g+=1;
}

for(int d = 1; d < rank; d++) gSwapHalo(phi, mpiInfo, d);
```

Listing A.4: GS-RB with if-tests

A.3 GS-RB 3D without if tests

```
/************************
 *  Red Pass
 ************************/
//Odd layers - Odd Rows
g = nGhostLayers[1]*sizeProd[1] +
    nGhostLayers[2]*sizeProd[2] +
    nGhostLayers[3]*sizeProd[3];
loopRedBlack3D(rhoVal, phiVal, sizeProd, trueSize,
    kEdgeInc, lEdgeInc,
        g, gj, gjj, gk, gkk, gl, gll);

//Odd layers - Even Rows
g = (nGhostLayers[1]+1)*sizeProd[1] +
    (nGhostLayers[2]+1)*sizeProd[2] +
    nGhostLayers[3]*sizeProd[3];
loopRedBlack3D(rhoVal, phiVal, sizeProd, trueSize,
    kEdgeInc, lEdgeInc,
        g, gj, gjj, gk, gkk, gl, gll);

//Even layers - Odd Rows
g = (nGhostLayers[1])*sizeProd[1] +
    (nGhostLayers[2])*sizeProd[2] +
    (nGhostLayers[3]+1)*sizeProd[3];
loopRedBlack3D(rhoVal, phiVal, sizeProd, trueSize,
    kEdgeInc, lEdgeInc,
        g, gj, gjj, gk, gkk, gl, gll);

//Even layers - Even Rows
g = (nGhostLayers[1] + 1)*sizeProd[1] +
    (nGhostLayers[2]+1)*sizeProd[2] +
    (nGhostLayers[3]+1)*sizeProd[3];
loopRedBlack3D(rhoVal, phiVal, sizeProd, trueSize,
    kEdgeInc, lEdgeInc,
        g, gj, gjj, gk, gkk, gl, gll);

for(int d = 1; d < rank; d++) gSwapHalo(phi, mpiInfo, d);
```

Listing A.5: main routine

```
  inline static void loopRedBlack3D(double *rhoVal,double
     *phiVal,long int *sizeProd, int *trueSize, int
     kEdgeInc, int lEdgeInc,
        long int g, long int gj, long int gjj, long int gk,
           long int gkk, long int gl, long int gll){

  gj = g + sizeProd[1];
  gjj= g - sizeProd[1];
```

```
    gk = g + sizeProd[2];
    gkk= g - sizeProd[2];
    gl = g + sizeProd[3];
    gll= g - sizeProd[3];

    for(int l = 0; l<trueSize[3]; l+=2){
      for(int k = 0; k < trueSize[2]; k+=2){
        for(int j = 0; j < trueSize[1]; j+=2){
          // msg(STATUS, "g=%d", g);
          phiVal[g] = 0.125*(phiVal[gj] + phiVal[gjj] +
                  phiVal[gk] + phiVal[gkk] +
                  phiVal[gl] + phiVal[gll] + rhoVal[g]);
          g +=2;
          gj  +=2;
          gjj +=2;
          gk  +=2;
          gkk +=2;
          gl  +=2;
          gll +=2;
        }
      g +=kEdgeInc;
      gj  +=kEdgeInc;
      gjj +=kEdgeInc;
      gk  +=kEdgeInc;
      gkk +=kEdgeInc;
      gl  +=kEdgeInc;
      gll +=kEdgeInc;
      }
    g +=lEdgeInc;
    gj  +=lEdgeInc;
    gjj +=lEdgeInc;
    gk  +=lEdgeInc;
    gkk +=lEdgeInc;
    gl  +=lEdgeInc;
    gll +=lEdgeInc;
    }

    return;
  }
```

Listing A.6: loop routine