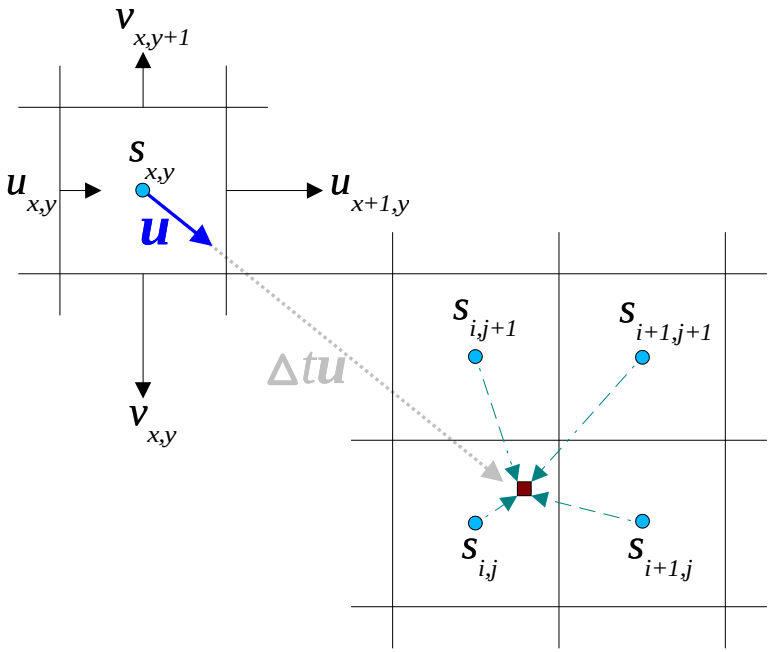
Semi-Lagrangian Method

- Review method



- MAC grid:

```
floats u, v, rho;
streamShape(u,   2, GRIDSIZE+1, GRIDSIZE);
streamShape(v,   2, GRIDSIZE,   GRIDSIZE+1);
streamShape(rho, 2, GRIDSIZE,   GRIDSIZE);
```

- If bound Δt, can get an effective stencil (bounds on how far we might have to look). `STENCIL` could be around 5-30.

Semi-Lagrangian Method: sample code

```
kernel void AdvectScalar(floats s[x:-STENCIL..STENCIL][y:-STENCIL..STENCIL],
                         floats u[x:0..1][y], floats v[x][y:0..1],
                         out floats snext[x][y], double dt, int gridsize) {
  // Get velocity at center of cell by averaging
  vec2 centervel = 0.5 * vec2(u[0][0] + u[1][0], v[0][0] + v[0][1]);

  // Get position in [0,1]x[0,1], backtrack, and map back into grid index
  vec2 pos = (vec2(x, y) + vec2(0.5, 0.5)) / gridsize;
  pos -= dt * centervel;
  pos = gridsize * pos - vec2(0.5, 0.5);

  // Get index for lower left cell that will be used in the interpolation
  int i = (pos[VX] > 0) ? (int)pos[VX] : (int)(pos[VX]-1.0);
  int j = (pos[VY] > 0) ? (int)pos[VY] : (int)(pos[VY]-1.0);

  // Interpolate values from four adjacent cells
  float s00 = GetScalarWithBoundary(i,   j,   s, gridsize);
  float s01 = GetScalarWithBoundary(i,   j+1, s, gridsize);
  float s10 = GetScalarWithBoundary(i+1, j,   s, gridsize);
  float s11 = GetScalarWithBoundary(i+1, j+1, s, gridsize);
  snext[0][0] = bilerp(pos[VX]-i, pos[VY]-j, s00, s01, s10, s11); }
```

Semi-Lagrangian Method: code issues

- All of the `[x][y]`'s in the argument list are somewhat confusing.
  - *e.g.* How do these kernels get executed?
    - `kernel void foo(floats a, floats b)`
    - `kernel void foo(floats a[x], floats b)`
    - `kernel void foo(floats a[x], floats b[y])`
    - `kernel void foo(floats a[x], floats b[x])`
      - What if a and b are streams of *different* lengths?
      - I can think of a case in which I would need it called as long as *one* of the streams has elements remaining (advection of velocity).

```
kernel void AdvectVelocityU(
  floats u[x:-STENCIL..STENCIL][y:-STENCIL..STENCIL],
  floats v[x:-1..0][y:0..1], out floats unext[x][y])
```

- What happens if you reference a value outside specified neighbor range?
  - *e.g.* Declare `floats a[x:-1..1]` and reference `a[2]`.
- What would be the syntax if we didn't have a stencil and wanted to reference arbitrary stream values inside a kernel?

Boundary Conditions

- I want `GetScalarWithBoundary` to look like

```
float GetScalarValueWithBoundary(int i, int j,
 floats s[x:-STENCIL..STENCIL][y:-STENCIL..STENCIL], int gridsize) {

  if (0 <= i && i < gridsize && 0 <= j && j < gridsize) {
    assert(abs(i-x) <= STENCIL && abs(j-y) <= STENCIL);
    return s[i-x][j-y];
  } else {
    // TODO: handle different boundary condition; return 0 for now
    return 0.0;
  } }
```

- This isn't really a kernel. I want `s` to be passed as-is from `AdvectScalar`.
- Eventually could handle more complicated boundary conditions in this function.
- A possible way around all of this would be to surround the grid with boundary cells (boundary of width `STENCIL`) which get initialized with the desired boundary values. This way wouldn't have to worry about accessing out-of-bounds element, and could just use

  ```
  float s00 = s[i-x][j-y];
  ```

  instead of calling `GetScalarValueWithBoundary`.

Banded Matrix Storage

- Would like to be able to declare a struct such as

```
struct BandedMatrix {
  floats diag, leftdiag, rightdiag, left, right;
  int offset;
};
```

  but currently structs containing streams are not supported.

Banded Matrix - Vector Multiplication

```
floats BandedMatrixVectorMult(BandedMatrix A, floats x) {
  int len = streamGetLength(x);

  floats b = VectorMult(x, A.diag);

  floats xoffset = streamDomain(x, A.offset,len);
  b = VectorAdd(b, VectorMult(xoffset, A.right));

  floats xoff1 = streamDomain(x, 1, len);
  b = VectorAdd(b, VectorMult(xoff1, A.rightdiag));

  floats boff1 = streamDomain(b, 1, len);
  boff1 = VectorAdd(boff1, VectorMult(x, A.leftdiag));

  floats boffset = streamDomain(b, A.offset, len);
  boffset = VectorAdd(boffset, VectorMult(x, A.left));

  return b;
}
```

- Requires appropriate padding with 0’s

Preconditioned Conjugate Gradient (PCG)

```
void PCG(BandedMatrix A, floats b, out floats x, double tol) {
  int i_max = 1000, i = 0; floats r, d;

  r = VectorSub(b, BandedMatrixVectorMult(A, x)); /* r = b − Ax */
  PreconditionerSolve(A, d, r); /* d = M^{-1} r */

  float delta_old, delta_new = VectorDot(r, d), delta_0 = delta_new;

  while (i < i_max && delta_new > tol*tol*delta_0) {
    floats q = BandedMatrixVectorMult(A, d); /* q = Ad */
    float alpha = delta_new / VectorDot(d, q);

    x = VectorAddScalarV(x, alpha, d); /* x+ = αd */
    r = VectorAddScalarV(r, -alpha, q); /* r− = αq */
    PreconditionerSolve(A, s, r); /* s = M^{-1} r */

    delta_old = delta_new;
    delta_new = VectorDot(r, s);
    float beta = delta_new/delta_old;
    d = VectorAddScalarV(r, beta, d); /* d = r + βd */

    i++; } }
```

PCG (cont'd)

- Ideally would like to be able to pass `PreconditionerSolve` to `PCG` as a callback function so that we can easily experiment with different preconditioners
  - So metacompiler should support passing pointers to functions (not kernels) taking streams. *e.g.*

    ```
    void foo(void (*cb)(floats a)) { floats b; cb(b); }
    ```
- **Preconditioners**:
  - Probably shouldn't use incomplete cholesky because it is harder to parallelize than other preconditioners
  - Currently implemented a simple multi-step Jacobi preconditioner (uses simple matrix and vector operations)
  - Might investigate multigrid preconditioners

More Questions/Comments

- Would like to be able to move stuff out of main() into separate functions (functions taking streams).
- Need to figure out return type of a function that returns a stream (already under discussion)
  - A related issue is how the stream data returned by the function should be declared?
    - New QuickSpec suggests `streamAlloc` which means kernel body `{ return *a + *b; }` needs to be rewritten

```
{ stream float *t = streamAlloc(sizeof(float));
  *t = *a + *b;
  return t; }
```

- What happens with

```
kernel void foo(floats a, out floats b) { *b = 0; *b = *a; }
```

  if it's called with the same stream: `foo(a,a)`? Can the programmer check for this case in the kernel (by comparing pointer values `a,b`)?

- What's the syntax and behavior for a kernel calling a kernel? *e.g.* is this valid

```
kernel void foo(floats a) { ... }
kernel void bar(floats a) { foo(a); }
```

More Questions/Comments (cont'd)

- Can a function be declared as `reduce`? How would you write the body of such function?
- Unclear about `REDUCE(func, &p, s)` example in new QuickSpec.
- Add `inout` keyword? *e.g.* currently writing the following using the `reduce` keyword, which seems weird

```
kernel void plusequals(inout floats a, floats b)
{ *a += *b; }
```