```
typedef double density;
typedef double distance;
typedef double filling;

class Smooth {
public:
  Smooth(int sizex = 100, int sizey = 100, distance inner = 21.0, filling birth_1 = 0.278,
         filling birth_2 = 0.365, filling death_1 = 0.267, filling death_2 = 0.445,
         filling smoothing_disk = 0.147, filling smoothing_ring = 0.028);
  int Size() const;
  int Sizex() const;
  int Sizey() const;
  int Range() const;
  int Frame() const;
  const std::vector<density> &Field() const;
  void Field(std::vector<density> const &input);

  //! \brief Piecewise linear function defining the disk
  //! \details
  //! - 1 inside the disk
  //! - 0 outside the disk
  //! - 0 < x < 1 in the smoothing region
  double Disk(distance radius) const;
  //! \brief Piecewise linear function defining the ring
  //! \details
  //! - 1 inside the ring
  //! - 0 outside the ring
  //! - 0 < x < 1 in the smoothing region
  double Ring(distance radius) const;
  //! Smooth step function: 0 at -∞, 1 at +∞
  static double Sigmoid(double variable, double center, double width);
  //! $e^{-4x / width}$: 0 at -∞, 1 at +∞
  static double Sigmoid(double x, double width);
  density Transition(filling disk, filling ring) const;
  int TorusDistance(int x1, int x2, int size) const;
  double Radius(int x1, int y1, int x2, int y2) const;
  // Value of the integral over a single ring
  double NormalisationRing() const;
  // Value of the integral over a single disk
  double NormalisationDisk() const;
  //! Sets the playing field to random values
  void SeedRandom();
  //! Sets the playing field to constant values
  void SeedConstant(density constant = 0);
  //! Adds a disk to the playing field
  void AddDisk(int x0 = 0, int y0 = 0);
  //! Adds a ring to the playing field
  void AddRing(int x0 = 0, int y0 = 0);
  //! Sets a single pixel in the field
  void AddPixel(int x0, int y0, density value);
  //! Moves to next step
  void Update();
  //! Prints current field to standard output
  void Write(std::ostream &out);
  //! Linear index from cartesian index
  int Index(int i  int j) const;
  //! Returns {disk, ring} integrals at point (x, y)
  std::pair<density, density> Integrals(int x, int y) const;
                ,
  //! Cartesian index from linear index
  std::pair<int, int> Index(int i) const;
private:
  int sizex, sizey;
  std::vector<density> field, work_field;
  filling birth_1, death_1;
  filling birth_2, death_2;
  filling smoothing_disk, smoothing_ring;
  distance inner, outer, smoothing;
  int frame;
  double normalisation_disk, normalisation_ring;
```

```
#ifdef HAS_MPI
public:
  MPI_Comm const &Communicator() const { return communicator; }
  void Communicator(MPI_Comm const &comm) { communicator = comm; }

  //! Figure start owned sites for given rank
  static int OwnedStart(int nsites, int ncomms, int rank);

  //! \brief Syncs fields between processes
  //! \details Assumes that each rank owns the sites given by OwnedRange.
  static void WholeFieldBlockingSync(std::vector<density> &field, MPI_Comm const &comm);

  //! Update which layers computation and communication
  void LayeredUpdate();

  //! \brief Syncs fields between processes without blocking
  //! \details Assumes that each rank owns the sites given by OwnedRange.
  static MPI_Request WholeFieldNonBlockingSync(std::vector<density> &field, MPI_Comm const &comm);
private:
  MPI_Comm communicator;
#endif
};
```

Tests

```
#include <cmath>
#include <random>
#include "catch.hpp"
#include "smooth.h"

TEST_CASE("Compute Integrals") {
  Smooth smooth(300, 300);
  smooth.SeedConstant(0);

  // check for different positions in the torus
  for(auto const x : {150, 298, 0})
    for(auto const y : {150, 298, 0}) {
      SECTION("At position (" + std::to_string(x) + ", " + std::to_string(y) + ")") {
        SECTION("Ring only") {
          smooth.AddRing(150, 150);

          auto const result = smooth.Integrals(150, 150);
          // 0.1 accuracy because of smoothing
          CHECK(std::get<0>(result) == Approx(0).epsilon(0.1));
          CHECK(std::get<1>(result) == Approx(1).epsilon(0.1));
        }

        SECTION("Disk only") {
          smooth.AddDisk(150, 150);
          auto const result = smooth.Integrals(150, 150);
          CHECK(std::get<0>(result) == Approx(1).epsilon(0.1));
          CHECK(std::get<1>(result) == Approx(0).epsilon(0.1));
        }

        SECTION("Disk and ring") {
          smooth.AddRing(150, 150);
          smooth.AddDisk(150, 150);
          auto const result = smooth.Integrals(150, 150);
          CHECK(std::get<0>(result) == Approx(1).epsilon(0.1));
          CHECK(std::get<1>(result) == Approx(1).epsilon(0.1));
        }
      }
    }
}

TEST_CASE("Update") {
  // just test playing with a single pixel lit up sufficiently that the
  // transition is non-zero in the ring.
  auto const radius = 5;
  Smooth smooth(100, 100, radius);
  smooth.AddPixel(50, 50, 0.3 * smooth.NormalisationRing());
  CHECK(std::get<0>(smooth.Integrals(50, 50))
        == Approx(0.3 * smooth.NormalisationRing() / smooth.NormalisationDisk()));

  // check the integrals are numbers for which Transition gives non-zero result
  // in the ring
  CHECK(std::get<1>(smooth.Integrals(50, 50)) == Approx(0));
  CHECK(std::get<0>(smooth.Integrals(40, 40)) == Approx(0));
  CHECK(std::get<1>(smooth.Integrals(40, 40)) == Approx(0.3));
  CHECK(std::get<0>(smooth.Integrals(42, 39)) == Approx(0));
  CHECK(std::get<1>(smooth.Integrals(42, 39)) == Approx(0.3));

  // Now call update
  smooth.Update();
  auto const field = smooth.Field();
  // And check death in the disk
  CHECK(field[smooth.Index(50, 50)] == Approx(0));
  CHECK(field[smooth.Index(51, 52)] == Approx(0));
  // And check life in the ring
  CHECK(field[smooth.Index(45, 45)] == Approx(smooth.Transition(0, 0.3)));
  CHECK(field[smooth.Index(42, 39)] == Approx(smooth.Transition(0, 0.3)));
  // And check death outside
  CHECK(field[smooth.Index(15, 15)] == Approx(0));
}
```

Tests

```
TEST_CASE("Arithmetics for plitting a field on different nodes") {
  CHECK(Smooth::OwnedStart(5, 2, 0) == 0);
  CHECK(Smooth::OwnedStart(5, 2, 1) == 3);

  for(int i(0); i < 5; ++i)
    CHECK(Smooth::OwnedStart(5, 5, i) == i);

  // with too many procs, some procs have empty ranges
  for(int i(5); i < 10; ++i)
    CHECK(Smooth::OwnedStart(5, 10, i) == 5);
}

TEST_CASE("Sync whole field") {
  int rank, ncomms;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &ncomms);

  // Create known field: -1 outside owned range, equal to rank inside
  // Different on each process!
  // Also, we make sure the size does not split evenly with the number of procs,
  // because that is a harder test.
  std::vector<density> field(5 * ncomms + ncomms / 3, -1);
  std::fill(field.begin() + Smooth::OwnedStart(field.size(), ncomms, rank),
            field.begin() + Smooth::OwnedStart(field.size(), ncomms, rank + 1), rank);

  SECTION("Blocking synchronisation") {
    Smooth::WholeFieldBlockingSync(field, MPI_COMM_WORLD);

    for(int r(0); r < ncomms; ++r)
      CHECK(std::all_of(field.begin() + Smooth::OwnedStart(field.size(), ncomms, r),
                        field.begin() + Smooth::OwnedStart(field.size(), ncomms, r + 1),
                        [r](density d) { return std::abs(d - r) < 1e-8; }));
  }

  SECTION("Non blocking synchronisation") {
    auto request = Smooth::WholeFieldNonBlockingSync(field, MPI_COMM_WORLD);
    MPI_Wait(&request, MPI_STATUS_IGNORE);

    for(int r(0); r < ncomms; ++r)
      CHECK(std::all_of(field.begin() + Smooth::OwnedStart(field.size(), ncomms, r),
                        field.begin() + Smooth::OwnedStart(field.size(), ncomms, r + 1),
                        [r](density d) { return std::abs(d - r) < 1e-8; }));
  }
}
```

Tests

```
TEST_CASE("Serial vs parallel") {
  Smooth serial(100, 100, 5);
  Smooth parallel(100, 100, 5);
  parallel.Communicator(MPI_COMM_WORLD);

  // generate one field for all Smooth instances
  std::vector<density> field(100 * 100);
  std::random_device rd; // Will be used to obtain a seed for the random number engine
  std::mt19937 gen(rd());
  std::uniform_real_distribution<> randdist(0, 1);
  std::generate(field.begin(), field.end(), [&randdist, &gen]() { return randdist(gen); });
  MPI_Bcast(field.data(), field.size(), MPI_DOUBLE, 0, MPI_COMM_WORLD);

  // set the fields for both Smooth instances
  serial.Field(field);
  parallel.Field(field);

  int rank, ncomms;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &ncomms);

  auto const start = Smooth::OwnedStart(field.size(), ncomms, rank);
  auto const end = Smooth::OwnedStart(field.size(), ncomms, rank);
  // if this is false, then the test itself is wrong
  CHECK(std::equal(serial.Field().begin() + start, serial.Field().begin() + end,
                   parallel.Field().begin() + start));

  SECTION("Blocking synchronization") {
    // check fields are the same in parallel and in serial for a few iterations
    for(int i(0); i < 3; ++i) {
      serial.Update();
      parallel.Update();
      CHECK(std::equal(serial.Field().begin() + start, serial.Field().begin() + end,
                       parallel.Field().begin() + start));
    }
  }

  SECTION("Layered communication-computation") {
    for(int i(0); i < 3; ++i) {
      serial.Update();
      parallel.LayeredUpdate();
      CHECK(std::equal(serial.Field().begin() + start, serial.Field().begin() + end,
                       parallel.Field().begin() + start));
    }
  }
}
```

Implementation

```
#include <cassert>
#include <cmath>
#include <cstdlib>
#include <iostream>

#include "smooth.h"

Smooth::Smooth(int sizex, int sizey, distance inner, filling birth_1, filling birth_2,
               filling death_1, filling death_2, filling smoothing_disk, filling smoothing_ring)
    : sizex(sizex), sizey(sizey), field(sizex * sizey), work_field(sizex * sizey), inner(inner),
      birth_1(birth_1), birth_2(birth_2), death_1(death_1), death_2(death_2),
      smoothing_disk(smoothing_disk), smoothing_ring(smoothing_ring), outer(inner * 3),
      smoothing(1.0)
#ifdef HAS_MPI
      ,
      communicator(MPI_COMM_SELF)
#endif
{
  normalisation_disk = NormalisationDisk();
  normalisation_ring = NormalisationRing();
}

const std::vector<density> &Smooth::Field() const { return field; };
void Smooth::Field(std::vector<density> const &input) {
  assert(field.size() == input.size());
  field = input;
}

int Smooth::Range() const { return outer + smoothing / 2; }

int Smooth::Sizex() const { return sizex; }
int Smooth::Sizey() const { return sizey; }
int Smooth::Size() const { return sizex * sizey; }

/// "Disk_Smoothing"
double Smooth::Disk(distance radius) const {
  if(radius > inner + smoothing / 2)
    return 0.0;
  if(radius < inner - smoothing / 2)
    return 1.0;
  return (inner + smoothing / 2 - radius) / smoothing;
}
/// end

double Smooth::Ring(distance radius) const {
  if(radius < inner - smoothing / 2)
    return 0.0;
  if(radius < inner + smoothing / 2)
    return (radius + smoothing / 2 - inner) / smoothing;
  if(radius < outer - smoothing / 2)
    return 1.0;
  if(radius < outer + smoothing / 2)
    return (outer + smoothing / 2 - radius) / smoothing;
  return 0.0;
}

double Smooth::Sigmoid(double variable, double center, double width) {
  return Sigmoid(variable - center, width);
}
double Smooth::Sigmoid(double x, double width) { return 1.0 / (1.0 + std::exp(-4.0 * x / width)); }

density Smooth::Transition(filling disk, filling ring) const {
  auto const sdisk = Sigmoid(disk - 0.5, smoothing_disk);
  auto const t1 = birth_1 * (1.0 - sdisk) + death_1 * sdisk;
  auto const t2 = birth_2 * (1.0 - sdisk) + death_2 * sdisk;
  return Sigmoid(ring - t1, smoothing_ring) * Sigmoid(t2 - ring, smoothing_ring);
}

int Smooth::Index(int i, int j) const { return i * Sizex() + j; }
std::pair<int, int> Smooth::Index(int i) const { return {i / Sizex(), i % Sizex()}; }
```

Implementation

```
int Smooth::TorusDistance(int x1, int x2, int size) const {
  auto const remainder = std::abs(x1 - x2) % size;
  return std::min(remainder, std::abs(remainder - size));
}


double Smooth::Radius(int x1, int y1, int x2, int y2) const {
  int xdiff = TorusDistance(x1, x2, sizex);
  int ydiff = TorusDistance(y1, y2, sizey);
  return std::sqrt(xdiff * xdiff + ydiff * ydiff);
}

double Smooth::NormalisationDisk() const {
  double total = 0.0;
  for(int x = 0; x < sizex; x++)
    for(int y = 0; y < sizey; y++)
      total += Disk(Radius(0, 0, x, y));
  return total;
}

double Smooth::NormalisationRing() const {
  double total = 0.0;
  for(int x = 0; x < sizex; x++)
    for(int y = 0; y < sizey; y++)
      total += Ring(Radius(0, 0, x, y));
  return total;
}
void Smooth::SeedRandom() {
  for(int x = 0; x < sizex; x++)
    for(int y = 0; y < sizey; y++)
      field[Index(x, y)] += (static_cast<double>(rand()) / static_cast<double>(RAND_MAX));
}

void Smooth::SeedConstant(density constant) { std::fill(field.begin(), field.end(), constant); }
void Smooth::AddDisk(int x0, int y0) {
  for(int x = 0; x < sizex; x++)
    for(int y = 0; y < sizey; y++)
      field[Index(x, y)] += Disk(Radius(x0, y0, x, y));
}

void Smooth::AddRing(int x0, int y0) {
  for(int x = 0; x < sizex; x++)
    for(int y = 0; y < sizey; y++)
      field[Index(x, y)] += Ring(Radius(x0, y0, x, y));
}

void Smooth::AddPixel(int x0, int y0, density value) { field[Index(x0, y0)] = value; }

void Smooth::Write(std::ostream &out) {
  for(int x = 0; x < sizex; x++) {
    for(int y = 0; y < sizey; y++)
      out << field[Index(x, y)] << " , ";
    out << std::endl;
  }
  out << std::endl;
}

int Smooth::Frame() const { return frame; }
```

```
void Smooth::Update() {
```

```
int rank, ncomms;
MPI_Comm_rank(Communicator(), &rank);
MPI_Comm_size(Communicator(), &ncomms);
```

```
WholeFieldBlockingSync(field, communicator);
```

```
 auto const start = OwnedStart(Size(), ncomms, rank);
 auto const end = OwnedStart(Size(), ncomms, rank + 1);
```

```
auto const start = 0;
auto const end = field.size();
```

```
for(int i(start); i < end; ++i)
```

```
auto const xy = Index(i);
auto const integrals = Integrals(xy.first, xy.second);
```

```
work_field[i] = Transition(integrals.first, integrals.second);
```

```
std::swap(field, work_field);
```

```
std::pair<density, density> Smooth::Integrals(int x, int y) const {
  density ring_total(0), disk_total(0);
  for(std::vector<density>::size_type i(0); i < field.size(); ++i)
```

```
auto const cartesian = Index(i);
int deltax = TorusDistance(x, cartesian.first, sizex);
if(deltax > outer + smoothing / 2)
  continue;
```

```
int deltay = TorusDistance(y, cartesian.second, sizey);
if(deltay > outer + smoothing / 2)
  continue;
```

```
double radius = std::sqrt(deltax * deltax + deltay * deltay);
double fieldv = field[i];
ring_total += fieldv * Ring(radius);
disk_total += fieldv * Disk(radius);
```

```
  return {disk_total / NormalisationDisk(), ring_total / NormalisationRing()};
}
```

```
int Smooth::OwnedStart(int nsites, int ncomms, int rank) {
  assert(nsites >= 0);
  assert(ncomms > 0);
  assert(rank >= 0 and rank <= ncomms);
```

```
  return rank * (nsites / ncomms) +
}
```

```
std::min(nsites % ncomms, rank);
```

```
void Smooth::WholeFieldBlockingSync(std::vector<density> &field, MPI_Comm const &comm) {
```

```
int rank, ncomms;
MPI_Comm_rank(comm, &rank);
MPI_Comm_size(comm, &ncomms);
```

```
if(ncomms == 1)
  return;
```

```
std::vector<int> displacements{0}, sizes;
```

```
for(int i(0); i < ncomms; ++i) {
  displacements.push_back(Smooth::OwnedStart(field.size(), ncomms, i + 1));
  sizes.push_back(displacements.back() - displacements[i]);
}
```

```
MPI_Allgatherv(MPI_IN_PLACE, ??[rank], MPI_DOUBLE, ?? .data(), ?? .data(),
                          ?? .data(), MPI_DOUBLE, comm);
```

```
MPI_Request Smooth::WholeFieldNonBlockingSync(std::vector<density> &field, MPI_Comm const &comm) {
```

```
int rank, ncomms;
MPI_Comm_rank(comm, &rank);
MPI_Comm_size(comm, &ncomms);

std::vector<int> displacements{0}, sizes;
```

```
for(int i(0); i < ncomms; ++i) {
  displacements.push_back(Smooth::OwnedStart(field.size(), ncomms, i + 1));
  sizes.push_back(displacements.back() - displacements[i]);
}
```

```
MPI_Request request;
MPI_Iallgatherv(MPI_IN_PLACE, ?? [rank], MPI_DOUBLE, ?? .data(), ?? .data(),
                          ??  .data(), MPI_DOUBLE, comm, & ??  );
```

```
  return request;
}
```

```
void Smooth::LayeredUpdate() {
```

```
int rank, ncomms;
MPI_Comm_rank(Communicator(), &rank);
MPI_Comm_size(Communicator(), &ncomms);
```

```
auto request = WholeFieldNonBlockingSync(field, communicator);
```

```
auto const start = OwnedStart(Size(), ncomms, rank);
auto const end = OwnedStart(Size(), ncomms, rank + 1);
auto const interaction = Sizex() * static_cast<int>(std::floor(outer + smoothing / 2 + 1));

auto const set_work_field_at_index = [this](int i) {
  auto const xy = Index(i);
  auto const integrals = Integrals(xy.first, xy.second);
  work_field[i] = Transition(integrals.first, integrals.second);
};
```

```
for(int i(start + interaction); i < end - interaction; ++i)
  set_work_field_at_index(i);
```

```
MPI_Wait(&request, MPI_STATUS_IGNORE);
```

```
for(int i(start); i < std::min(end, start + interaction); ++i)
  set_work_field_at_index(i);
```

```
for(int i(std::min(end, end - interaction)); i < end; ++i)
  set_work_field_at_index(i);
```

```
std::swap(field, work_field);
```