Exercise Sheet No.2 Deadline: 05. November 2013, 2:00 pm

To install the libconfig library and the files mentioned in the exercise you can use the script 'installer.sh' provided on the homepage. Simply place the two archives in the same location. libconfig will be installed into a folder 'Software' in your home directory. If you want to change this adapt the file, e.g. by removing the install instruction or changing the destination. Please note that you then have to change the path in the Makefile as well. Build the provided project files by typing "make".
Please send your solutions to **ole.klein@iwr.uni-heidelberg.de** before the deadline.

EXERCISE 2 EFFICIENCY VERSUS SOFTWARE DESIGN

In this exercise you will write an efficient two dimensional convolution and compare its performance to a highly modular and object oriented implementation. The latter uses an iterator based grid implementation of a structured rectangular grid which frequently occur in finite element software packages. Our implementation was optimized for cell centered finite volume methods and consists of the classes `Grid`, `ElementIterator`, `Element`, `FaceIterator` and `Face`. A detailed description of the `Grid` interface is appended.

In real-life partial differential equations problems we usually have to deal with spatially varying parameters. A very comfortable implementation of a parameter class which allows the input of image (greyscale TIFF or native CImg) files in two and three dimensions is given by the `SpatialParameters` class provided on the lecture homepage. Additionally, it allows a mapping from the TIFF byte values to `double` values (even `double` vectors of arbitrary size). A detailed description of the `SpatialParameters` interface is appended. The implementation of this class depends on the open-source libraries `libconfig` and `CImg`.

In this exercise you will implement a two dimensional convolution of a function $f(\vec{x})$ by a kernel $k(\vec{x})$ with

$$k(\vec{x}) = -\sigma^3 \pi \Delta G(\vec{x}) \tag{1}$$

where Δ denotes the laplace operator and $G(\vec{x})$ denotes the normalized Gaussian in two dimensions

$$G(\vec{x}) = \frac{1}{2\pi\sigma} \exp\left\{ -\frac{||x||^2}{2\sigma^2} \right\}. \tag{2}$$

Hence, we compute

$$\mathcal{C}f(\vec{x}) = \int_\Omega f(\vec{z}) \;\; k(\vec{x} - \vec{z}) \; d^2z. \tag{3}$$

For the convolution of an image, we assume that $f(\vec{x})$ is piecewise constant (on each pixel). On each pixel i, we denote the constant value with f_i and the subdomain related to the pixel with $P_i \subset \Omega$. Using Gauss' divergence theorem (in 2d), we may write

$$\begin{aligned} \mathcal{C}f(\vec{x}) &= \sum_i \int_{P_i} f_i \;\; k(\vec{x} - \vec{z}) \; d^2z \\ &= \sum_i \int_{\partial P_i} f_i \;\; \vec{n} \cdot \vec{\nabla}(-\sigma^3 \pi G(\vec{x} - \vec{z})) \; ds(\vec{z}). \end{aligned} \tag{4}$$

An implementation of the convolution as given by (4) is provided on the homepage in the file `convolution.cc` together with an example image `startimage.tif` and a mapping for the TIFF byte values in `image.cfg`.

1.) Study the implementation to learn the correct usage of the `Grid` and `SpatialParameters` class. (You may also take a look at the source code, but you do not have to study it in detail). Notice

the extensive use of local iterator objects.

2.) Write a more efficient implementation of (4) without refering to the `Grid` class (You can use `fast_draft.cc` as template). Compare its runtime for the test image with the other given implementation.

To obtain a fair comparison you should compile like

```
g++ convolution.cc -o convolution -O2 -DNDEBUG -Wall -lconfig++ -lX11
```

as the implementation uses `assert` macros which should be deactivated by `-DNDEBUG` and optimization is mandatory when checking performance.

Appendix A - The Grid Interface

Cell centered finite volume discretizations are usually based on a partition of the spatial domain given by a structured rectangular grid. An implementation of such a grid was realized using four different classes. The idea is as follows:

We declare a grid class `Grid` which provides iterators of type `ElementIterator` to access its elements. The iterators may be dereferenced to objects of an element class `Element` which provides iterators of type `FaceIterator` to access its faces. Those may be dereferenced to objects of a face class `Face`.

The element and face objects are encapsulated in the corresponding iterators which are therefore declared as `friend` (see listing at the end). The geometric information provided by the element and face objects (e.g. volume and area) should be precomputed when the corresponding iterator is incremented (or created) such that they may be accessed repeatedly at little computational cost.

The interface defines the following public member functions:

For the grid:

- **Extent of grid**
  `const Vector & Grid::extent() const:`
  Returns a reference to a vector holding the width, height (and length) of the grid.
- **Number of cells**
  `const std::vector<size_t> & Grid::cells() const:`
  Returns the number of grid cells in each of the dimensions.
- **Constructor**
  `Grid::Grid(const Vector _extent, const std::vector<size_t> _cells):`
  Constructor which sets the extent of the grid and the number of cells in each grid dimension.
- **Start iterator**
  `ElementIterator Grid::begin() const:`
  Returns an iterator to the first grid element.
- **End iterator**
  `ElementIterator Grid::end() const:`
  Returns an invalid iterator reached by incrementing an iterator to the last grid element.

For the element iterator:

- **Increment (in-place) operator**
  `ElementIterator & ElementIterator::operator++():`
  Increments the iterator and triggers the computation of the return values of `Element::barycenter()`, `Element::extent()`, and `Element::volume()` for the encapsulated element object.

- **Reset iterator**
  `ElementIterator & ElementIterator::reset():`
  Resets the iterator to the first grid element and triggers the computation of the return values of `Element::barycenter()`, `Element::extent()` and `Element::volume()` for the encapsulated element object.

- **Dereference operator**
  `Element & ElementIterator::operator*():`
  Returns a reference to the encapsulated object.

- **Negative Comparison**
  `bool ElementIterator::operator!=(const ElementIterator & it) const:`
  Checks whether two iterators point to different grid elements.

- **Element index**
  `const size_t & ElementIterator::id() const:`
  Returns a unique consecutive index for the current grid element beginning with zero for the first element.

For the element:

- **Position of barycenter**
  `const Vector & Element::barycenter() const:`
  Returns the position of the element's barycenter.

- **Extent**
  `const Vector & Element::extent() const:`
  Returns the width, height (and length) of the element.

- **Volume**
  `const double & Element::volume() const:`
  Returns the volume of the element

- **Start iterator**
  `FaceIterator Element::begin() const:`
  Returns an iterator to the first element face.

- **End iterator**
  `FaceIterator Element::end() const:`
  Returns an invalid iterator reached by incrementing an iterator to the last element face.

For the face iterator:

- **Increment (in-place) operator**
  `FaceIterator & FaceIterator::operator++():`
  Increments the iterator and triggers the computation of the return values of `Face::area()`, `Face::dist_to_barycenter()`, `Face::normal_component()`, `Face::normal_direction()` and `Face::is_boundary()` for the encapsulated face object.

- **Reset iterator**
  `FaceIterator & FaceIterator::reset():`
  Resets the iterator to the first element face and triggers the computation of the return values of `Face::area()`, `Face::dist_to_barycenter()`, `Face::normal_component()`, `Face::normal_direction()` and `Face::is_boundary()` for the encapsulated face object.

- **Dereference operator**
  `Face & FaceIterator::operator*():`
  Returns a reference to the encapsulated object.

- **Negative Comparison**
  `bool FaceIterator::operator!=(const FaceIterator & it) const:`
  Checks whether two iterators point to different element faces.

- **Element index**
  `const size_t & FaceIterator::in_id() const:`
  Returns a unique consecutive index for the element on the inner side of this face (the element which provided this iterator).

- **Element index**
  `const size_t & FaceIterator::out_id() const:`
  Returns a unique consecutive index for the element on the outer side of this face.

For the face:

- **Area of the face**
  `const double & Face::area() const:`
  Returns the area of the face.

- **Distance to next barycenter**
  `const double &Face::dist_to_barycenter() const:`
  Returns the distance to the barycenter of the element on the outer side of the face.

- **Boundary flag**
  `const bool & Face::is_boundary() const:`
  Returns true, when this face is on the domain boundary.

- **Normal component**
  `unsigned int Face::normal_component() const:`
  Returns the dimension index of the non-zero component of the face's normal vector. The normal vector points to the element on the outer side of this face.

- **Normal direction**
  `int Face::normal_direction() const:`
  Returns the sign of the non-zero component of the face's normal vector. The normal vector points to the element on the outer side of this face.

As the grid was implemented for both the two and three dimensional case, we require the precompiler macro `DIMENSIONS` to be set. Notice, that a branching like `if(Grid::dimensions == 3)` will be resolved at compile time without introducing additional computational cost at run time.

Appendix B - The SpatialParameter Class Interface

The class `SpatialParameters` provides the following public interface:

- **Constructor**
  `const SpatialParameters::`
  `SpatialParameters(string filename, Grid & grid, string map_file = string(""),`
  `string map_name = string("")):`
  This should read the parameters from a file. It should accept both standard grayscale `.tiff` files as well as `CImg`'s native file format with the `.cimg` extension. The grid cell size is allowed to vary from the image pixel size by an integer multiple and an appropriate transformation from image pixels to grid cells will be performed automatically. A mapping of the TIFF byte values to floating point values may be provided in a configuration file `map_file`. As multiple mappings may be declared in the same file, the name of the mapping should be provided in `map_name`. See the example files given on the lecture homepage to get the syntax right. If the TIFF byte value in the mapping list is followed by more than one floating point value, the value is mapped to the corresponding multi dimensional vector.

- **Parameter value at coordinate**
  `const double & SpatialParameters::`
  `operator()(const Vector & position,`
  `const int p = 0) const:`
  Returns the value at the coordinates given by `position` (Values on element faces are undefined). If the mapping range is multi dimensional, `p` denotes the component to evaluate.

- **Parameter value at cell id**
  `const double & SpatialParameters::operator()(const size_t & id,`
  `const int p = 0) const:`
  Returns the value for the cell with the global index `id` within the grid provided by `ElementIterator::id()`. If the mapping range is multi dimensional, `p` denotes the component to evaluate.

- **Parameter value at cell id (for modification)**
  `double & SpatialParameters::operator()(const size_t & id,`
  `const int p = 0) :`
  Returns the value for the cell with the global index `id` within the grid provided by `ElementIterator::id()`. If the mapping range is multi dimensional, `p` denotes the component to evaluate. This method is not constant and should allow a lasting modification of the value in each grid cell (e.g. in a preprocessing step).

- **Dimension of Mapping Range**
  `int SpatialParameters::parameters() :`
  Returns the dimension of the mapping range.

- **ASCII Output**
  `void SpatialParameters::debug_ascii_output(int p = 0,`
  `usigned int z = 0) :`
  Draws an ASCII image of the parameter field of the range dimension `p`. If the parameter field is three dimensional, then `z` needs to hold the corresponding grid index indicating the two dimensional x-y slice to be drawn.

- **Image Output**
  `void SpatialParameters::save_as_image(const std::string filename,`
  `int p = 0) :`
  Save the parameter field of range dimension `p` to a file. The file extension should indicate which image format to use. The format must be supported by the `CImg` library and in case of three dimensions the resulting file may not be processed by standard image viewer applications.

5 Points