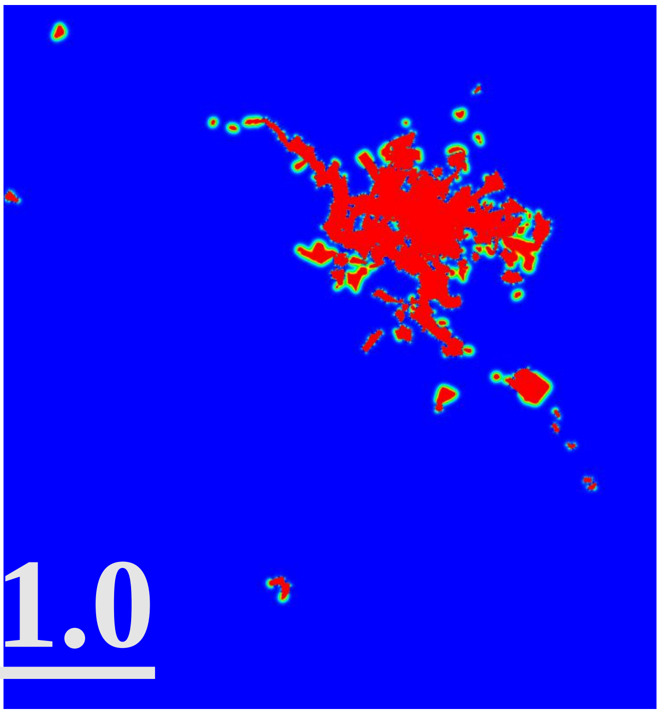


UFLow 1.0

Flow was developed in 2018/2019.

I can be reached at
koscianski => utfpr , edu , br

Introduction

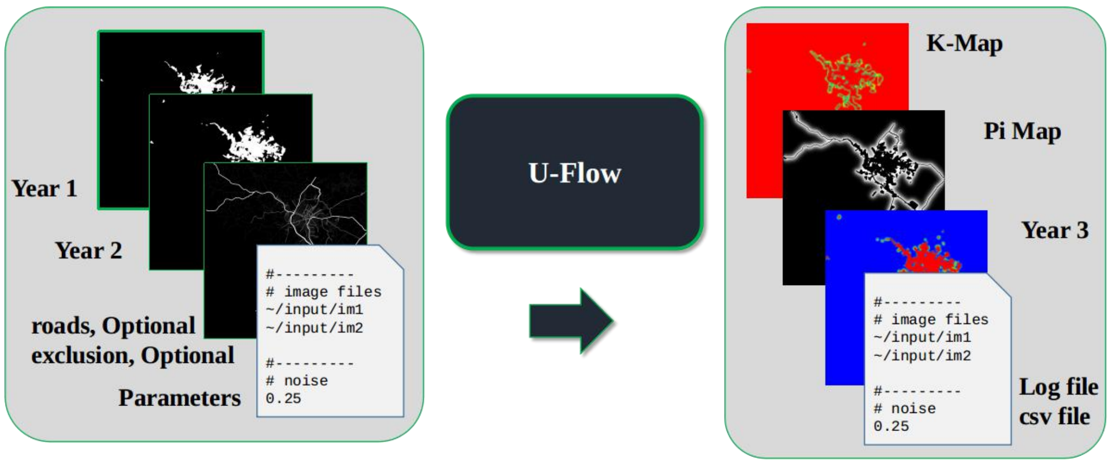
UFlow was originally conceived as a model for urban sprawl.

At its core, it takes a process of diffusion as a metaphore for the way a city expands. In short, “hot” areas tend to warm up their surroundings. Evidently this is far from representing the true dynamics of any urban agglomeration; but for any model to capture fine details, more data is necessary. Deciding at which point to cut the ambitions of a model in reproducing exactly what happens in the real world, is a problem on itself. UFlow does it pragmatically, pretending no more than detecting “hot” areas of a city and calculating projections based on that.

A second part of the model deals with (tries to) the emergence of city patches at random. A gas station. An industrial plant. An isolated commercial enterprise. Such projects and investments may take place at more or less isolated regions, disconnected from the main mass that compose a city. As it is done in other models, UFlow rolls a dice, according to certain parameters that will be described.

In a Glimpse

The essential input consists of two images of the city, taken at different times. An optional map of roads can be furnished. Parameters are set in a text file.



```
#---------
# image files
~/input/im1
~/input/im2

#---------
# noise
0.25
```

The model runs a calibration process, trying to determine the relative intensity of the urbanization process in different places accross the map. This procedure outputs two files: the “KMap” and the “PiMap”.

After calibration, forecast is computed. Besides some images written to disk, there’s a log file, a csv, some statistics shown in the screen.

All images in PNG format.

Installation

The code was built using g++ version 7 on Linux.

Just type “make”, that’s it. An executable will be generated; there’s no generation of shortcuts, menu items or anything else.

The code includes only one library:

LodePNG version 20180611

Copyright (c) 2005-2018 Lode Vandevenne

The kd-tree was based on Rosetta code; this function is disactivated.

Img companion tool

A small command line tool is provided; it performs simple operations with PNG files.

Simply type “make img”. Help is provided by typing “img”.

Input Images

All images in PNG format.

UFlow read the files and disregard color information; RGB channels are averaged, resulting a float in the range [0 ; 1].

Values greater than 0.5 mean “urban”. Using this rule, the images are internally regarded as binary.

The same applies to the optional Road Map.

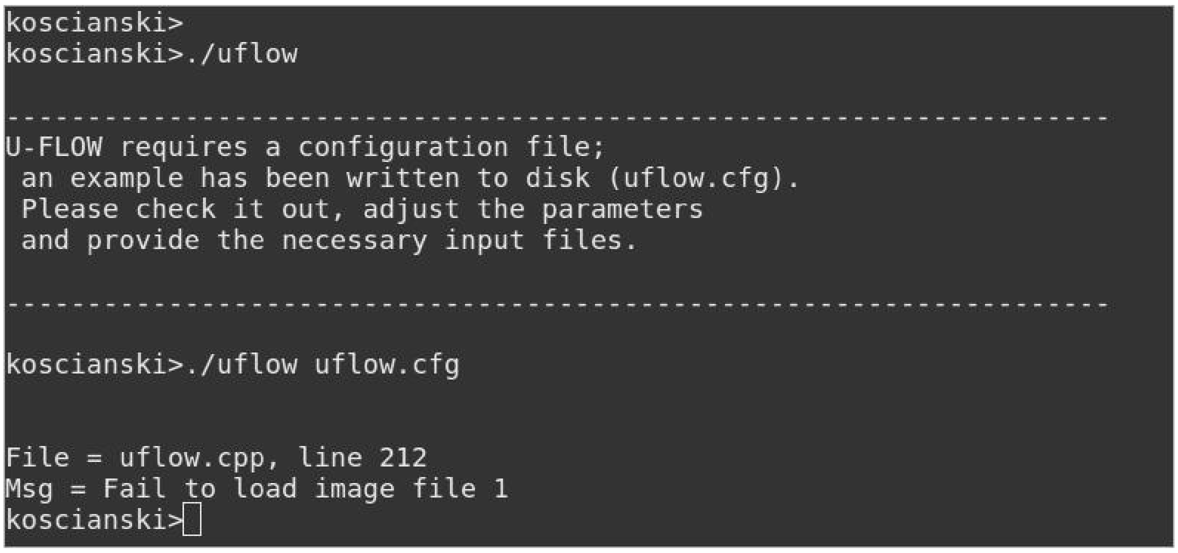
An also optional background image can be furnished. It will be used only to produce the final output.

	Image	**Description**
required	urban footprint 1	white pixel = urban
required	urban footprint 2	white pixel = urban
	roads	white pixel = road
	background	color, only for output

Image Resolution: tests were performed with images ranging from ~ 200 x 200 up to 1200 x 1200.

Basic Configuration File

The executable can provide a template.
Just run it:



```
koscianski>
koscianski>./uflow

--------------------------------------------------------------------
U-FLOW requires a configuration file;
 an example has been written to disk (uflow.cfg).
 Please check it out, adjust the parameters
 and provide the necessary input files.

--------------------------------------------------------------------

koscianski>./uflow uflow.cfg


File = uflow.cpp, line 212
Msg = Fail to load image file 1
koscianski>
```

Essential parameters are described in the following tables.

The config file is Case Sensitive.

Line in cfg file	**Meaning (all of these are images)**

image1 = path
image2 = path
	Urban footprints.

roads	Optional, map of roads.
exclusion	Optional. Black pixels will never be urbanized.
PiMap	Calibration output, provide path+filename
SaveKMap	Calibration output, provide path+filename
backgroundImg	Optional, self-explanatory
simulationImg	Calibration output, provide path+filename
simulationErr	Calibration output, provide path+filename
forecastImg	Forecast image.

Line in cfg file	**Meaning**
SimulationIsRepeatable = YES	The random generator always starts with the same value. If set to NO, the generator takes the system time as its starting point.
SkipCalibration = NO	Set to YES if the file KMap.png was already computed, and you want to test changes to other parameters.
deltaTime = 3.0	Time between urban footprints (image 1 and 2). A float number. The actual unit (be it days, years..) only have meaning to the user.

Line in cfg file	**Meaning**

DistanceMap=..
sigma=..
	These parameters are only used if you enable them in the source code.
	Not likely.

PiMap = ... (path)
HeatCyclesPiMap = 2000
	The PiMap is the intersection between image 2 and the road map.
	It is “heated”, causing the expansion of the urban footprint.
	The resulting image is a probability map, where random urban patches will be generated. Tune the value for your experiment empirically.

MaxCalibrationLoops = 20	Self-explanatory. Values between 20 and, perhaps, 50.
MaxHeatCycles = 5000	During each calibration loop, the heat equation may be iterated this much, not more. If you need/want to, change the value. Log files will guide you.
gamma = 0.08 0 < gamma < 1	At each calibration cycle, the simulator stacks a layer of insulation in regions that should not allow heat/flow to pass. Values around 0.0625 (2^{-4}) have worked fine.
KMapNoise = 0.3 0 < KMapNoise < 1	At the moment of the forecast, the diffusion parameter is perturbed with noise. This breaks isotherms and let some random urbanization to happen around the city.

Line in cfg file	**Meaning**
UrbanSprayForecast = YES	Turns on the random generation of urban fragments.
forecastTime = 3.0	This is the companion from parameter **deltaTime**. If your forecast repeats the time interval between image 1 and image 2, simply repeat the value here.

Hardwired Parameters

Some variables with a significant impact on the behavior of the model were set in the code and not exposed to users by means of configuration files, neither linked to the calibration process; their values were found empirically during the tests.

This opens two possibilities: (i) extending the configuration with more parameters; and (ii) experimenting with the automatic calibration of these additional variables.

Two values are defined in `"uflow.h"`

```
#define URBANTEMPERATURE 5.0
#define URBANTHRESHOLD   0.5
```

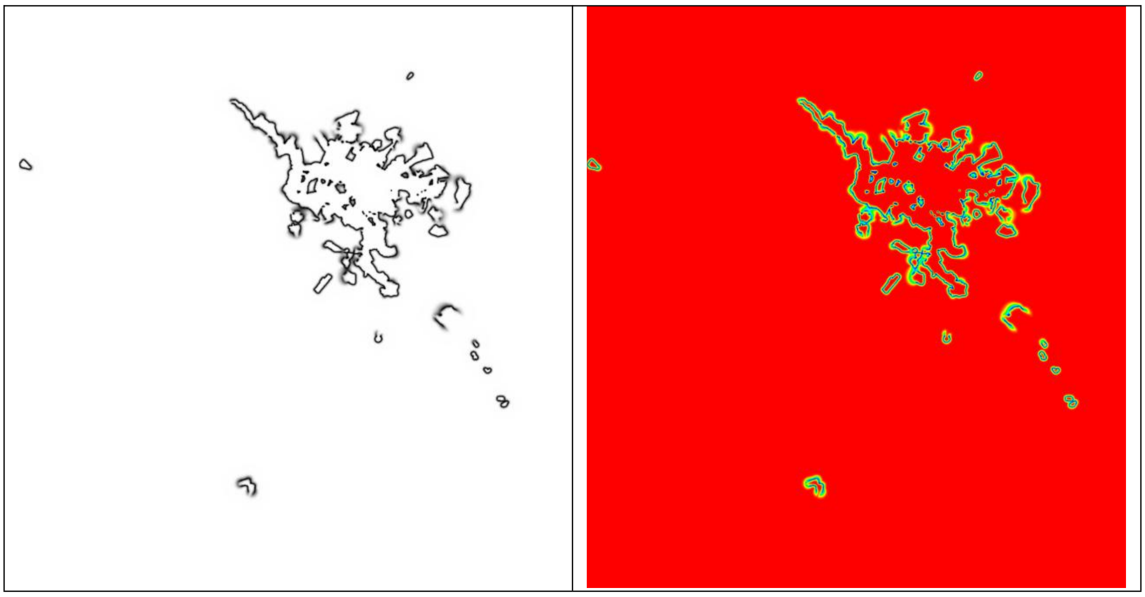
The ‘temperature’ corresponds to the value assigned to urban pixels before running the model.
The ‘threashold’ value is used to interpret input images, and to classify outputs. Any pixel with value greater than this constant will be handled as an urbanized cell.

Two other values control the numerical integration scheme: they are the space and the time steps.
Distance between nodes in the grid was set to 1.0.
The time step was set to 0.0625, or 2^{-4}. Modifying this value, as is the case with the ‘temperature controls’, has a direct impact on calibration results.

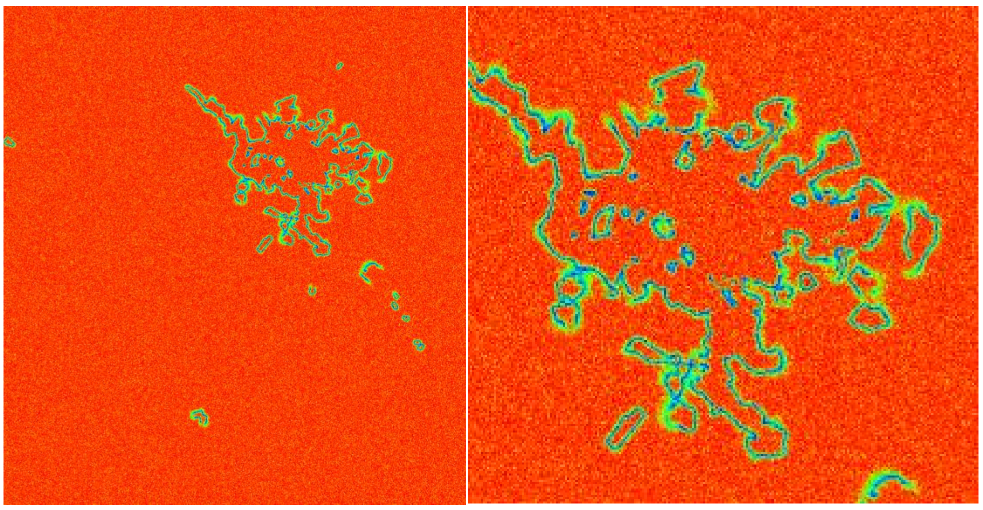
Output Images - calibration

The calibration of the simulation consists of computing a K-Map and a PiMap.

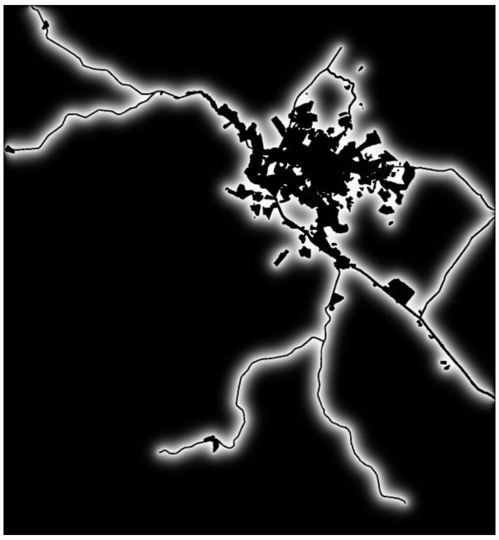
A typical KMap should resemble to this:



Blue regions are insulated (values near 0), red regions let the flow pass (values near 1.0).
After being perturbed by noise, the K-map looks this way:



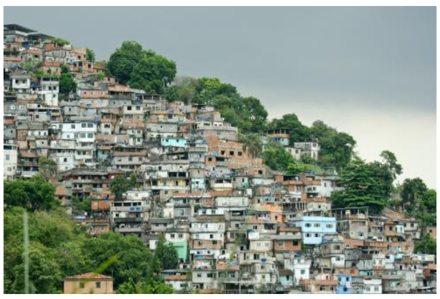
Here's an example of the "PiMap":



"Warmer" zones are those where random urbanization is more likely to occur. UFlow uses the same heuristic of many other simulators, meaning that urbanization is more likely to occur in the vicinity of urban zones (= city blocks and roads).

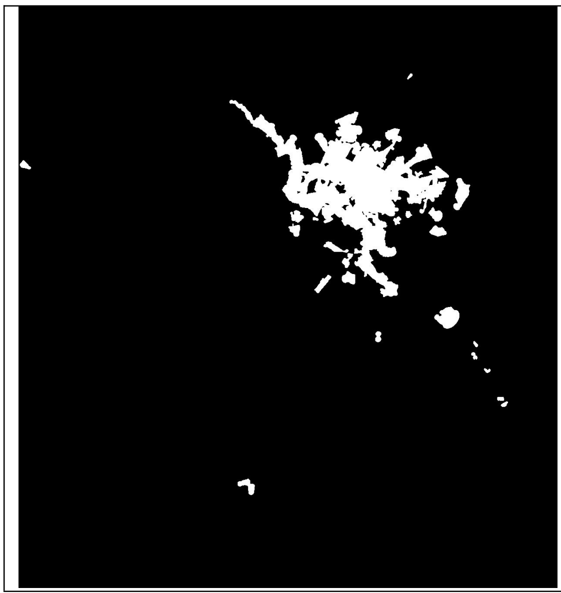
Remember: you can set the parameter
SkipCalibration = YES
and modify the KMap and the PiMap at will.

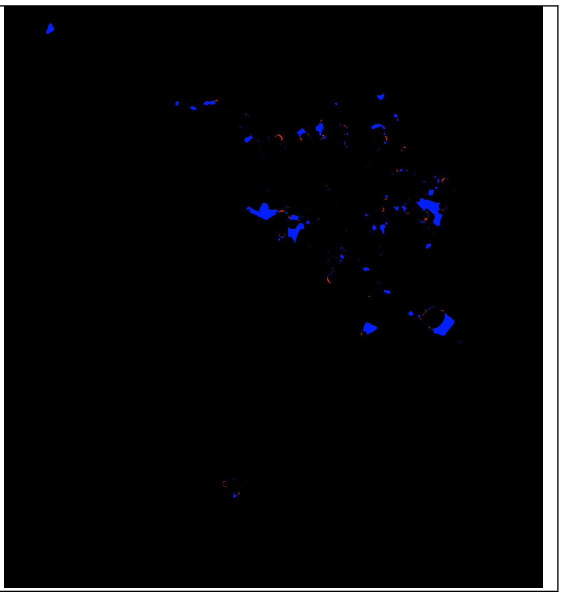
An expert might paint regions of those maps using information that cannot be deduced from the input images (urban footprints).



For instance, UFlow was designed disregarding slope, because in its home country, Brazil, a (very) steep hill was never enough to stop people from building irregular houses.

UFlow generates simulation images that are compared with the second true image to verify the accuracy of the calibration.



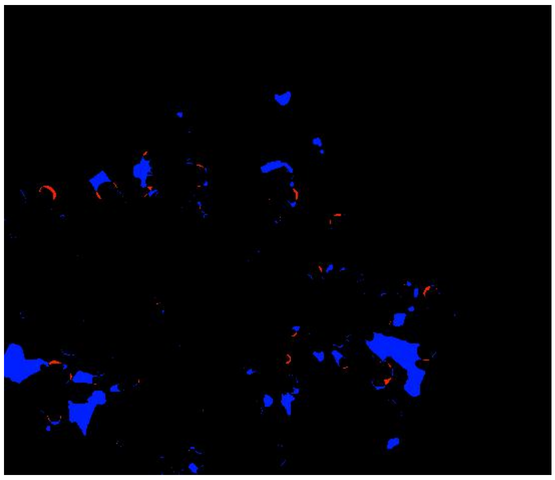


The colors of the pixels in the image to the right have the following meanings:

red = false positive. These pixels (urban blocks) were wrongly generated by the simulator.

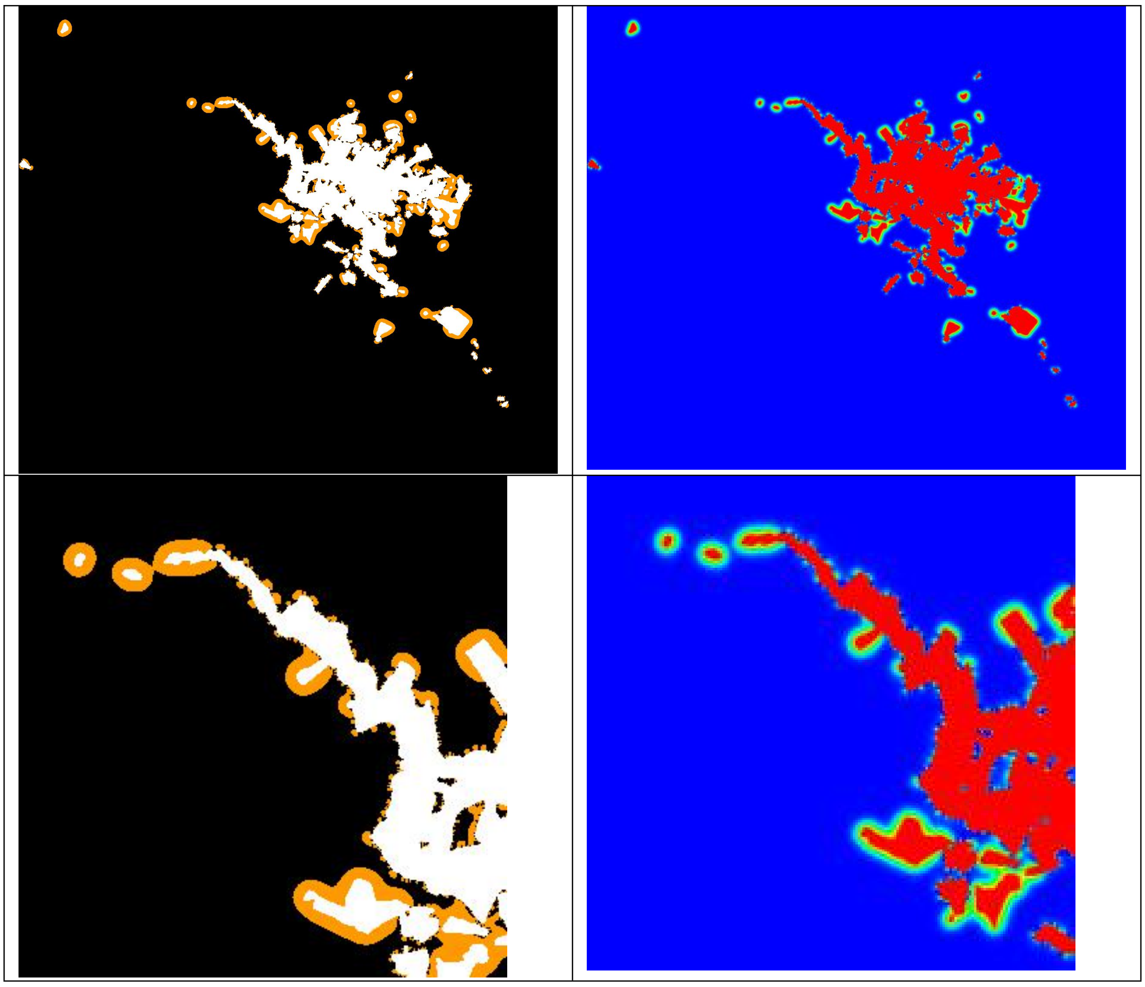
blue = false negative. Those areas were left empty by the generator.

A zoom of the previous image:



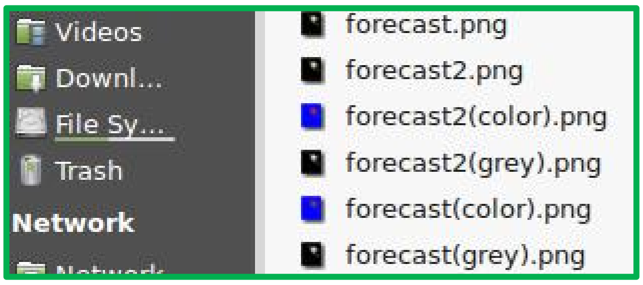
Output - forecast images

Here’s a forecast image, in two versions:



UFlow generates a few versions of output images. These different versions will be written using the base path and filename provided in the configuration.

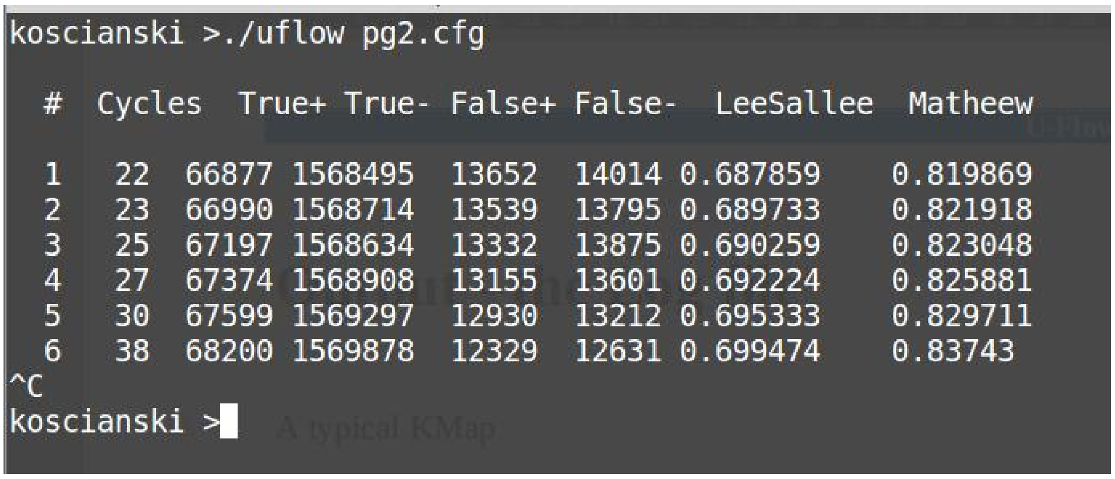
Example:



forecast.png
forecast2.png
forecast2(color).png
forecast2(grey).png
forecast(color).png
forecast(grey).png

Output - the CSV file

During execution the simulator shows statistics on the screen.



```
koscianski >./uflow pg2.cfg

  #  Cycles  True+ True-  False+ False-   LeeSallee   Matheew

  1   22  66877 1568495  13652  14014 0.687859    0.819869
  2   23  66990 1568714  13539  13795 0.689733    0.821918
  3   25  67197 1568634  13332  13875 0.690259    0.823048
  4   27  67374 1568908  13155  13601 0.692224    0.825881
  5   30  67599 1569297  12930  13212 0.695333    0.829711
  6   38  68200 1569878  12329  12631 0.699474    0.83743
^C
koscianski >
```

The first column is the count of calibration cycles; its max value is given by **MaxCalibrationLoops** in the config file.

Cycles: number of iterations of integration of the Heat Equation; its max value is given by **MaxHeatCycles**.

True/False +/- : count of pixels [not] correctly placed during calibration.

Lee-Sallee: metric. Comparison between image-2 and simulation. Printed on the screen, only for reference.

Matthews' coefficient: metric, same purpose as Lee-Sallee. This one is used by the simulator to decide when to stop calibration.

All these values are written to a file, format CSV, same name and path as the config file.

Output - the Log file

At the end of the simulation process, a log is written to disk.

After general information (as image sizes), the last part of the log looks like this:

```
   UFlow results
==================

Start Execution
Calibration time = 333.849 seconds
Forecast       time = 3.50265 seconds
Total Exec    time = 338.41 seconds

         ... (more info)

List of clusters, second image:

19 26 34 36 54 93 97 124 151 154
154 161 164 165 176 185 231 243 243 262
264 318 494 513 514 575 787 806 1011 1125
1127 1268 1279 1379 1388 1561 3479 6046 129207

Avg. cluster size = 26 (smaller ones)

         ... (more info)

Simulation: final metrics.
   Urban pixels = 71999 (+ 41.3 % over 1st image)
   Heat cycles    = 234
   Lee Sallee     = 0.857
   Matthews         = 0.937
      True    Positive / Negative = 71579 / 1582089
      False Positive / Negative = 420 / 8950

---------------------------------
           Forecast
---------------------------------

Time elapsed (from second image) = 9 years
Heat cycles                  = 73
Lin. regression          = 110089 (expected new city size)
Area new cluster        = 26
Number new clusters = 0
Expansion (new pixels):
  clusters                       = 0 (0% of total)
  city sprawl                 = 110277
Forecast city size    = 110277 (+ 36.9 % over 2nd image)
```

Creating new parameters

Adding new user parameters is a simple task.
Each parameter or variable must appear in one line of the configuration file; sequence is of no importance and white spaces are trimmed.
Variables are set using classic assignment syntax, e.g:

```
Xis = 3.1415
flag = TRUE
path = ./bla/ble/bli.txt
```

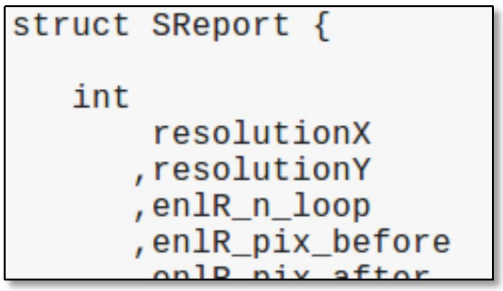
The config file is scanned once, and pairs tag/value are stored in memory. Values are accessed with helper functions:

```
double xis = config_getd ("Xis");
aflag = config_getb ("flag");
str = config_gets ("path");
```

Extending the report

The report system is entirely contained in the header file `"report.h"`.

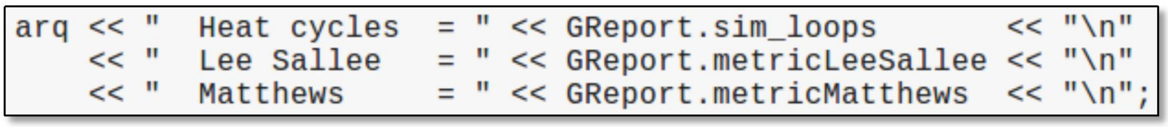
The structure SReport contains data fields that will be stored in the output ASCII file. Adding new fields is a matter of declaring them in the code:



```
struct SReport {

   int
       resolutionX
      ,resolutionY
      ,enlR_n_loop
      ,enlR_pix_before
       enlR_pix_after
```

Generating the report is also straightforward; a sample of the code is self-explanatory:



```
arq << "  Heat cycles    = " << GReport.sim_loops        << "\n"
    << "  Lee Sallee     = " << GReport.metricLeeSallee  << "\n"
    << "  Matthews       = " << GReport.metricMatthews   << "\n";
```

Inside the simulator, values are set to fields using the global variable GReport:

```
GReport.variable = value;
```

Generation of the report is found at the last lines of the code of the simulator,

```
GenerateReport ((str2 + ".log").c_str());
```