How to Write Perf Tests

Lets consider the following example:

```
using namespace std;
using namespace cv;
using namespace perf;

/* 1. Define parameter type and test fixture */
typedef std::tr1::tuple<Size, MatType, MatDepth> Size_MatType_OutMatDepth_t;
typedef perf::TestBaseWithParam<Size_MatType_OutMatDepth_t>
Size_MatType_OutMatDepth;

/* 2. Declare the testsuite */
PERF_TEST_P( Size_MatType_OutMatDepth, integral1,
    testing::Combine(
            testing::Values( TYPICAL_MAT_SIZES ),
            testing::Values( CV_8UC1, CV_8UC4 ),
            testing::Values( CV_32S, CV_32F, CV_64F ) ) )
{
    /* 3. Get actual test parameters */
    Size size = std::tr1::get<0>(GetParam());
    int matType = std::tr1::get<1>(GetParam());
    int sdepth = std::tr1::get<2>(GetParam());

    /* 4. Allocate and initialize arguments for tested function */
    Mat src(size, matType);
    Mat sum(size, sdepth);

    /* 5. Manifest your expectations about this test */
    declare.in(src, WARMUP_RNG).out(sum);

    /* 6. Collect the samples! */
    TEST_CYCLE(100) { integral(src, sum, sdepth); }

    /* 7. Add simple regression check */
    SANITY_CHECK(sum);
}
```

Define parameter type and test fixture

Each parameterized test requires a test fixture class to be able to accept parameters. We also recommend to create an alias for parameter type itself (using `typedef`). The recommended naming style for fixture class is just an enumeration of the argument types:

```
typedef std::tr1::tuple<cv::Size, perf::MatType, perf::MatDepth>
Size_MatType_OutMatDepth_t;
typedef perf::TestBaseWithParam<Size_MatType_OutMatDepth_t>
Size_MatType_OutMatDepth;
```

Performance testing framework provides a couple of useful wrappers for such constants as `Mat` type and `Mat` depth. Their names are `MatType` and `MatDepth`. Use them for your test fixture definitions to get well formatted values of test arguments in the final performance reports.

There are also two helper macro to define printable wrappers for any enumerations used as test parameters - `CV_ENUM` and `CV_FLAGS`. First one is intended for simple enumerations or sets of define and second should be used if members of enumeration are flags (can be combined with `|` operator)

Here is an usage example:

```
CV_ENUM (ReduceOp, CV_REDUCE_SUM, CV_REDUCE_AVG, CV_REDUCE_MAX, CV_REDUCE_MIN)
CV_FLAGS(NormType, NORM_INF, NORM_L1, NORM_L2, NORM_TYPE_MASK, NORM_RELATIVE,
NORM_MINMAX)
```

Declare the testsuite

Performance testing framework provides special macro to declare a parameterized performance tests:

```
PERF_TEST_P(fixture, name, params)
```

Declaration of value-parameterized performance test looks very close to googletest's `TEST_P` macro. However it is different and actually combines `TEST_P` and `INSTANTIATE_TEST_CASE_P` into a single construction.

- It is possible to use one test fixture for several test suites but each suite needs an unique name.
- name argument can be any valid C** identifier. Usually it contains name of tested function. - Into the params argument you can pass any list of parameters supported by googletest library. `TYPICAL_MAT_SIZES` used in our examples is a predefined list of most typical sizes of the image for performance tests. It is defined as (actual definition can be slightly different from this):

```
#define TYPICAL_MAT_SIZES ::perf::szVGA, ::perf::sz720p, ::perf::sz1080p,
::perf::szODD
```

Get actual test parameters

First block of well-written test should get all test parameters. Also all constants used in test body should be defined in this section.

If you need an input file for your test then you should use `getDataPath` method to get the actual location of the file. (This function uses `OPENCV_TEST_DATA_PATH` environment variable to get the location of test data.)

```
std::string cascadePath =
getDataPath("cv/cascadeandhog/cascades/lbpcascade_frontalface.xml");
```

If you need a temporary output file, then use `cv::tempfile()` function from core module to generate unique temporary path.

Allocate and initialize arguments for tested function

Second block inside the test is responsible for data initialization. Allocate and initialize all input data here and prepare structures for output of tested function.

Manifest your expectations about this test

This section is used to inform performance testing framework about your test. The entry point of all declarations is a declare member inherited from `::perf::TestBase` class.

The following options are available:

Input arguments declaration:

```
declare.in(arg1 [, arg2 [, arg3 [, arg4]]] [, warmup_type]);
```

Use this declaration to inform framework about input arguments of tested function. It will collect some info about them and this info will be available for further analysis. Method can accept 1-4 arguments of any type compatible with `cv::InputArray`, the last optional parameter is used to make a warmup/initialization of test data before the run. Available values are: `WARMUP_READ`, `WARMUP_WRITE`, `WARMUP_RNG`, `WARMUP_NONE`

Output argument declaration:

```
declare.out(arg1 [, arg2 [, arg3 [, arg4]]] [, warmup_type]);
```

Use this declaration to inform framework about output arguments of tested function. It will collect some info about them and this info will be available for further analysis. Method can accept 1-4 arguments of any type compatible with `cv::InputArray`, the last optional parameter is used to make a warmup/initialization of test data before the run. Values of this parameter are the same as in `declare.in` method. Desired number of iterations:

```
declare.iterations(n);
```

Use this declaration to specify how many samples you want to collect. Actual number of collected samples can be less then number set by this method, if test is reached time limit. Time limit:

```
declare.time(seconds);
```

This declaration can be used to specify time limit for the whole test run time. This is soft bound, so working function will not be interrupted even if it excess the limit but sampling will be stopped as soon as tested function returns. By default this limit is set to 3 seconds for desktop and to 6 seconds for Android devices. And it is possible to chain all these declarations:

```
declare.in(a, b, WARMUP_RNG).in(c).out(dst).time(0.5).iterations(100);
```

Collect the samples!

This is the heart of the test - main sampling loop.

There are three recommended ways for writing this loop:

Simplest loop - will iterate until the termination criteria is reached:

```
SIMPLE_TEST_CYCLE()
{
    threshold(src, dst, 100.0, 255.0, THRESH_BINARY);
}
```

Loop with desired iterations number:

```
TEST_CYCLE(100)
{
    threshold(src, dst, 100.0, 255.0, THRESH_BINARY);
}
```

Actually it is absolutely the same as:

```
declare.iterations(100);
SIMPLE_TEST_CYCLE()
{
    threshold(src, dst, 100.0, 255.0, THRESH_BINARY);
}
```

Custom loop where you can place some additional cleanup or initialization code on every iteration:

```
while(next())
{
res.clear();

startTimer();
{
    //only this block is measured
    cc.detectMultiScale(img, res, 1.1, 3, 0, minSize);
}
stopTimer();
}
```

Add a simple regression check

And final touch is adding simple regression check into your test:

```
SANITY_CHECK(sum [, epsilon]);
```

This command does not do full regression check but provides a fast and simple way to ensure that values returned by tested function are adequate.