
DSO: Direct Sparse Odometry

For more information see <https://vision.in.tum.de/dso>

1. Related Papers

- **Direct Sparse Odometry**, *J. Engel, V. Koltun, D. Cremers*, In arXiv:1607.02565, 2016
- **A Photometrically Calibrated Benchmark For Monocular Visual Odometry**, *J. Engel, V. Usenko, D. Cremers*, In arXiv:1607.02555, 2016

Get some datasets from <https://vision.in.tum.de/mono-dataset>.

2. Installation

```
1 git clone https://github.com/JakobEngel/dso.git
```

2.1 Required Dependencies

suitesparse and eigen3 (required). Required. Install with

```
1 sudo apt-get install libsuitesparse-dev libeigen3-dev libboost-all-dev
```

2.2 Optional Dependencies

OpenCV (highly recommended). Used to read / write / display images. OpenCV is **only** used in `IOWrapper/OpenCV/*`. Without OpenCV, respective dummy functions from `IOWrapper/*_dummy.cpp` will be compiled into the library, which do nothing. The main binary will not be created, since it is useless if it can't read the datasets from disk. Feel free to implement your own version of these functions with your preferred library, if you want to stay away from OpenCV.

Install with

```
1 sudo apt-get install libopencv-dev
```

Pangolin (highly recommended). Used for 3D visualization & the GUI. Pangolin is **only** used in `IOWrapper/Pangolin/*`. You can compile without Pangolin, however then there is not going to be any visualization / GUI capability. Feel free to implement your own version of `Output3DWrapper` with your preferred library, and use it instead of `PangolinDSOViewer`

Install from <https://github.com/stevenlovegrove/Pangolin>

zliblib (recommended). Used to read datasets with images as .zip, as e.g. in the TUM monoVO dataset. You can compile without this, however then you can only read images directly (i.e., have to unzip the dataset image archives before loading them).

```
1 sudo apt-get install zlib1g-dev
2 cd dso/thirdparty
3 tar -zxvf libzip-1.1.1.tar.gz
4 cd libzip-1.1.1/
5 ./configure
6 make
7 sudo make install
8 sudo cp lib/zipconf.h /usr/local/include/zipconf.h # (no idea why
   that is needed).
```

sse2neon (required for ARM builds). After cloning, just run `git submodule update --init` to include this. It translates Intel-native SSE functions to ARM-native NEON functions during the compilation process.

2.3 Build

```
1 cd dso
2 mkdir build
3 cd build
4 cmake ..
5 make -j4
```

this will compile a library `libdso.a`, which can be linked from external projects. It will also build a binary `dso_dataset`, to run DSO on datasets. However, for this OpenCV and Pangolin need to be installed.

3 Usage

Run on a dataset from <https://vision.in.tum.de/mono-dataset> using

```
1 bin/dso_dataset \
2   files=XXXXX/sequence_XX/images.zip \
3   calib=XXXXX/sequence_XX/camera.txt \
```

```
4 gamma=XXXXX/sequence_XX/pcalib.txt \  
5 vignette=XXXXX/sequence_XX/vignette.png \  
6 preset=0 \  
7 mode=0
```

See https://github.com/JakobEngel/dso_ros for a minimal example on how the library can be used from another project. It should be straight forward to implement extensions for other camera drivers, to use DSO interactively without ROS.

3.1 Dataset Format. The format assumed is that of <https://vision.in.tum.de/mono-dataset>. However, it should be easy to adapt it to your needs, if required. The binary is run with:

- `files=XXX` where XXX is either a folder or .zip archive containing images. They are sorted *alphabetically*. for .zip to work, need to compile with zlib support.
- `gamma=XXX` where XXX is a gamma calibration file, containing a single row with 256 values, mapping [0..255] to the respective irradiance value, i.e. containing the *discretized inverse response function*. See TUM monoVO dataset for an example.
- `vignette=XXX` where XXX is a monochrome 16bit or 8bit image containing the vignette as pixelwise attenuation factors. See TUM monoVO dataset for an example.
- `calib=XXX` where XXX is a geometric camera calibration file. See below.

Geometric Calibration File. Calibration File for Pre-Rectified Images

```
1 Pinhole fx fy cx cy 0  
2 in_width in_height  
3 "crop" / "full" / "none" / "fx fy cx cy 0"  
4 out_width out_height
```

Calibration File for FOV camera model:

```
1 FOV fx fy cx cy omega  
2 in_width in_height  
3 "crop" / "full" / "fx fy cx cy 0"  
4 out_width out_height
```

Calibration File for Radio-Tangential camera model

```
1 RadTan fx fy cx cy k1 k2 r1 r2  
2 in_width in_height  
3 "crop" / "full" / "fx fy cx cy 0"  
4 out_width out_height
```

Calibration File for Equidistant camera model

```
1 EquiDistant fx fy cx cy k1 k2 k3 k4
2 in_width in_height
3 "crop" / "full" / "fx fy cx cy 0"
4 out_width out_height
```

(note: for backwards-compatibility, “Pinhole”, “FOV” and “RadTan” can be omitted). See the respective `::distortCoordinates` implementation in `Undistorter.cpp` for the exact corresponding projection function. Furthermore, it should be straight-forward to implement other camera models.

Explanation: Across all models `fx fy cx cy` denotes the focal length / principal point **relative to the image width / height**, i.e., DSO computes the camera matrix K as

```
1 K(0,0) = width * fx
2 K(1,1) = height * fy
3 K(0,2) = width * cx - 0.5
4 K(1,2) = height * cy - 0.5
```

For backwards-compatibility, if the given `cx` and `cy` are larger than 1, DSO assumes all four parameters to directly be the entries of K , and ommits the above computation.

That strange “0.5” offset: Internally, DSO uses the convention that the pixel at integer position (1,1) in the image, i.e. the pixel in the second row and second column, contains the integral over the continuous image function from (0.5,0.5) to (1.5,1.5), i.e., approximates a “point-sample” of the continuous image functions at (1.0, 1.0). In turn, there seems to be no unifying convention across calibration toolboxes whether the pixel at integer position (1,1) contains the integral over (0.5,0.5) to (1.5,1.5), or the integral over (1,1) to (2,2). The above conversion assumes that the given calibration in the calibration file uses the latter convention, and thus applies the -0.5 correction. Note that this also is taken into account when creating the scale-pyramid (see `globalCalib.cpp`).

Rectification modes: For image rectification, DSO either supports rectification to a user-defined pinhole model (`fx fy cx cy 0`), or has an option to automatically crop the image to the maximal rectangular, well-defined region (`crop`). `full` will preserve the full original field of view and is mainly meant for debugging - it will create black borders in undefined image regions, which DSO does NOT ignore (i.e., this option will generate additional outliers along those borders, and corrupt the scale-pyramid).

3.2 Commandline Options there are many command line options available, see `main_dso_pangolin.cpp`. some examples include - `mode=X`: - `mode=0` use iff a photometric calibration exists (e.g. TUM monoVO dataset). - `mode=1` use iff NO photometric calibration exists (e.g. ETH EuRoC MAV dataset). - `mode=2` use iff images are not photometrically distorted (e.g. synthetic datasets).

- `preset=X`

-
- `preset=0`: default settings (2k pts etc.), not enforcing real-time execution
 - `preset=1`: default settings (2k pts etc.), enforcing 1x real-time execution
 - `preset=2`: fast settings (800 pts etc.), not enforcing real-time execution. WARNING: overwrites image resolution with 424 x 320.
 - `preset=3`: fast settings (800 pts etc.), enforcing 5x real-time execution. WARNING: overwrites image resolution with 424 x 320.
-
- `nolog=1`: disable logging of eigenvalues etc. (good for performance)
 - `reverse=1`: play sequence in reverse
 - `nogui=1`: disable gui (good for performance)
 - `nomt=1`: single-threaded execution
 - `prefetch=1`: load into memory & rectify all images before running DSO.
 - `start=X`: start at frame X
 - `end=X`: end at frame X
 - `speed=X`: force execution at X times real-time speed (0 = not enforcing real-time)
 - `save=1`: save lots of images for video creation
 - `quiet=1`: disable most console output (good for performance)
 - `sampleoutput=1`: register a “SampleOutputWrapper”, printing some sample output data to the commandline. meant as example.

3.3 Runtime Options Some parameters can be reconfigured from the Pangolin GUI at runtime. Feel free to add more.

3.4 Accessing Data. The easiest way to access the Data (poses, pointclouds, etc.) computed by DSO (in real-time) is to create your own `Output3DWrapper`, and add it to the system, i.e., to `FullSystem.outputWrapper`. The respective member functions will be called on various occasions (e.g., when a new KF is created, when a new frame is tracked, etc.), exposing the relevant data.

See `IOWrapper/Output3DWrapper.h` for a description of the different callbacks available, and some basic notes on where to find which data in the used classes. See `IOWrapper/OutputWrapper/SampleOutputWrapper.h` for an example implementation, which just prints some example data to the commandline (use the options `sampleoutput=1` `quiet=1` to see the result).

Note that these callbacks block the respective DSO thread, thus expensive computations should not be performed in the callbacks, a better practice is to just copy over / publish / output the data you need.

Per default, `dso_dataset` writes all keyframe poses to a file `result.txt` at the end of a sequence, using the TUM RGB-D / TUM monoVO format ([timestamp x y z qx qy qz qw] of the cameraToWorld transformation).

3.5 Notes

- the initializer is very slow, and does not work very reliably. Maybe replace by your own way to get an initialization.
- see https://github.com/JakobEngel/dso_ros for a minimal example project on how to use the library with your own input / output procedures.
- see `settings.cpp` for a LOT of settings parameters. Most of which you shouldn't touch.
- `setGlobalCalib(...)` needs to be called once before anything is initialized, and globally sets the camera intrinsics and video resolution for convenience. probably not the most portable way of doing this though.

4 General Notes for Good Results

Accurate Geometric Calibration

- Please have a look at Chapter 4.3 from the DSO paper, in particular Figure 20 (Geometric Noise). Direct approaches suffer a LOT from bad geometric calibrations: Geometric distortions of 1.5 pixel already reduce the accuracy by factor 10.
- **Do not use a rolling shutter camera**, the geometric distortions from a rolling shutter camera are huge. Even for high frame-rates (over 60fps).
- Note that the reprojection RMSE reported by most calibration tools is the reprojection RMSE on the “training data”, i.e., overfitted to the the images you used for calibration. If it is low, that does not imply that your calibration is good, you may just have used insufficient images.
- try different camera / distortion models, not all lenses can be modelled by all models.

Photometric Calibration Use a photometric calibration (e.g. using https://github.com/tum-vision/mono_dataset_code).

Translation vs. Rotation DSO cannot do magic: if you rotate the camera too much without translation, it will fail. Since it is a pure visual odometry, it cannot recover by re-localizing, or track through strong rotations by using previously triangulated geometry.... everything that leaves the field of view is marginalized immediately.

Computation Speed If your computer is slow, try to use “fast” settings. Or run DSO on a dataset, without enforcing real-time.

Initialization The current initializer is not very good... it is very slow and occasionally fails. Make sure, the initial camera motion is slow and “nice” (i.e., a lot of translation and little rotation) during initialization. Possibly replace by your own initializer.

5 License

DSO was developed at the Technical University of Munich and Intel. The open-source version is licensed under the GNU General Public License Version 3 (GPLv3). For commercial purposes, we also offer a professional version, see <http://vision.in.tum.de/dso> for details.