polyjam

Introduction and Operating Manual

polyjam is a C++ library for setting up algebraic geometry problems and generating efficient C++ code that solves the underlying polynomial systems of equations. It notably does so by applying the theory of Groebner bases. The solution to systems of multi-variate polynomial equations may be required in many engineering disciplines, which is why the tools provided through this library are likely to be of broad utility. The library is published under the GPL license. Please contact the author for use in proprietary applications.

Introduction to Groebner bases

The following is only a lay summary of the underlying theory, for more information the reader is kindly referred to Cox et al. 2007 (*Ideals, Varieties, and Algorithms*).

A set of polynomial equations defines a so-called *variety*, meaning a set of loci where all equations vanish (can also be empty). An example of a variety in 2D would for instance be an ellipse, which is defined by a single equation in two variables and of order 2. In the general case, two planes in 3D form a variety that is given by a line. It is interesting to note that the variety is not uniquely defined by a set of equations. For instance, in the last example, any two planes that intersect in the line could for instance define the variety. A set of equations that defines the variety is called a *basis*. There are an infinite number of bases that define the same variety, and we can typically generate new equations in the basis (so-called *generators*) by combining the existing equations, for instance linearly in the case of two planes in 3D. In order to capture this idea, we introduce the idea of an *ideal*, which is the set of all equations that can be formed by combining the generators of a basis, and, as such, the set of all equations that vanish on the variety. A variety is therefore defined by an ideal, and not just a set of equations, and a basis notably is a basis of the ideal.

The type of problem we are interested in here is the *minimal case*. It differs from the above examples by the fact that the defined varieties are no longer given by an infinite number of points. In fact, the ellipse and the line are one-dimensional varieties as we can parameterise all points on the variety by introducing a single free parameter. In consequence, they contain an infinite number of points. In the minimal case, the solution collapses down to a finite set of points, which we then call a zero-dimensional variety. An example would for instance be the intersection of two ellipses in the plane, which—in the general case—would result in a variety of four points. Note that even here, we can generate alternative ideal bases by a combination of the two equations. It is important to define what such a combination in the case of multi-variate polynomial equations actually is, as it is not necessarily only linear. **A polynomial combination is the formation of a new polynomial by taking two existing polynomials, multiplying each one by an arbitrary, individual term or monomial, and summing them up.** It is easily verified that the result is a new polynomial that still vanishes on the variety.

Let us at last move over to the *Groebner basis*. A Groebner basis is a special ideal basis that can—just like any other ideal basis—be generated by sequential combination of existing equations in the basis (according to the definition of a polynomial combination given above). However, a Groebner basis fulfils special properties. For instance, according to its definition, the generators of a Groebner basis are able to divide any member of the ideal with zero rest using multi-variate polynomial division and a certain (fixed) monomial ordering. The Groebner basis furthermore represents a special form that allows us to easily derive all solutions to a certain problem in the minimal case. We can furthermore find a Groebner basis by executing a relatively simple algorithm (called

Buchberger's algorithm), which is guaranteed to succeed.

The above is probably unclear without further reading on the Groebner basis theory, but—fortunately—understanding it is not necessary for grasping the essentials of efficient solver generation. The only important thing to know is that **finding a Groebner basis (which is sort of a good basis) is a process that requires iterative generation of new polynomials through polynomial combination until certain termination criteria are met**.

The art of efficient solver generation

At this point one may say "Wow that's great. All we need to do is chase a set of equations through Buchberger's algorithm, which will give us the Groebner basis, and—due to the good properties of a Groebner basis—also easily the solution to our problem". Unfortunately the matter is a little more complicated, and this is where most people get confused. So let's try to get this right.

We may indeed follow the above procedure and simply apply the Groebner basis solver of Maple or any other CAS to derive our Groebner basis directly from the set of equations of an instance of a particular problem. However, there are two problems associated with this:

- The search for a Groebner basis is somewhat exhaustive. The method typically has to investigate a very large number of polynomial combinations of generators before it can terminate. For problems that have to be solved many times, this represents a serious issue.
- The form of the generators of the final Groebner basis depends on the cancellation of terms alongside the derivation of the intermediate polynomials (due to vanishing coefficients). However, with limited machine precision and very long, exhaustive computations, the detection of vanishing coefficients becomes a serious challenge, if not impossible.

However, there is a fortunate fact that we can exploit in order to provide a remedy to this problem: **The sequence of polynomial combinations that has to be investigated and the form of all intermediate results is typically invariant for a general type of problem**. This means that, despite of coefficients that depend on the particular instance of a problem, the monomials appearing in the intermediate and final polynomials typically remain the same. Furthermore, **many of the polynomial combinations that have to be investigated in order to suffice the existence of a Groebner basis are in fact unnecessary for the generation of the final equations**.

This has a very important consequence: We have to search the Groebner basis only once! Once we have found the Groebner basis, we check which polynomial combinations are really necessary in order to arrive at the final equations. In other words, from a single Groebner basis search, we can extract a much shorter recipe that we can apply almost blindly to any instance of the same problem (different coefficients, but same form) in order to arrive at the generators of the Groebner basis. Furthermore, we have the luxury of choosing an instance with integer coefficients for the expensive on-time Groebner basis computation, thus eliminating the problem of limited machine precision.

In summary, the art of efficient solver generation consists of finding a compact recipe for computing the final Groebner basis generators from a single problem instance (defined in a finite field), and translating this recipe into a C++ routine. Since this recipe is applicable to any instance of the same problem, we have obtained an efficient (minimal) solver. **This is what polyjam is all about!**

How does polyjam work?

Early versions of polyjam did exactly what is described in the previous paragraph. It ran

Buchberger's algorithm to compute the Groebner basis for a finite field instance of a certain problem. It then back-traced the polynomial combinations that were effectively used to form the final polynomials, and translated this recipe into C++ code that could then be applied to any instance of the problem (including those with real coefficients).

The latest version of polyjam works differently. It applies the method presented in Kukelova et al. 2008 (*Automatic Generator of Minimal Problem Solvers*), which consists of a multi-step procedure: We first compute the Groebner basis (for instance using Buchberger's algorithms, but more modern methods are of course fine as well). All we keep from this initial computation is the form of the final polynomials (the so-called basis monomials). This part is done using an external Groebner basis solver generator (i.e. Macaulay2). In a next step, we generate a lot of new polynomials by multiplying the original polynomials with all monomials up to a certain degree, store all finite coefficients of all resulting equations in a large matrix (grouped such that each column contains the coefficients corresponding to a certain monomial), and perform Gauss-elimination. We repeat this procedure with an increasing degree until all final equations are obtained. Note that Gauss elimination on the large matrix sort of replaces the polynomial combinations in Buchberger's method. The outlined procedure results again in too many polynomials, so the recipe is completed by eliminating unnecessary polynomials one-by-one, each time checking whether Gauss elimination on the remaining matrix still leads to all required final polynomials. The advantage of Kukelova's method is that the recipe simply translates into copying the coefficients into a large matrix, and performing Gauss-elimination on the latter, for which there exist numerically stable solvers.

Generating a solver for a certain type of problem therefore is a five-step procedure:

1) Create an instance of a problem with integer coefficients in polyjam.
2) Call Macaulay2 to compute the Groebner basis for this problem instance, and communicate the form of this basis to polyjam.
3) Multiply the original equations by all monomials up to a certain degree, store the coefficients in a matrix (where each column represents a monomial), and perform Gauss-elimination. Repeat this procedure with increasing degree until all required polynomials are found.
4) Remove unnecessary lines in the matrix one-by-one.
5) Translate the obtained elimination template into C++ code (i.e. a piece of code that will fill a matrix with coefficients of the original equations depending on the result).

Note that, in order to complete the solver such that it produces the finite number of solutions, some more routines need to be added in order to extract the actual solutions from the final Groebner basis. Polyjam uses Eigenvalue decomposition of an *Action matrix* for solving this problem, and automatically adds this to the code. The Action matrix is a multi-variate extension to the so-called *Companion matrix*, but a detailed discussion of this would go beyond the scope of this introduction.

It is true that the art of automatic solver generation is somewhat black, as understanding the exact role of the above five steps is non-trivial, and information on the subject is often fragmented, incomplete, or provided in a cryptical, mathematical language. The above summary may help in understanding the basic steps of solver generation from a practical perspective, but certainly does not go beyond. The most important message at this point is that polyjam takes care of steps 2 to 5, and all you have to do in order to solve a certain problem is to define the initial form of the problem.

Installing the library

The library has been tested under Linux and OSX, and the installation instructions for these two systems do not really differ. There are a few dependencies which have to be installed upfront:

-*OpenCV*: The library uses OpenCV for optional visualisation of the elimination template. Use a standard package manager to install OpenCV (such as *brew* under OSX).
-*Macaulay2*: The library uses Macaulay2 for the initial Groebner basis computations. Please visit the page http://www.math.uiuc.edu/Macaulay2/ to obtain OS-specific installation instructions.

Once these dependencies are installed, we can move over to polyjam itself. Either download the package from https://github.com/laurentkneip/polyjam or clone it using git:

```
git clone https://github.com/laurentkneip/polyjam
```

There are four folders in the main path:

1) *polyjam_documentation*: Contains the present document.
2) *polyjam_generator*: Contains the core library for automatic solver generation.
3) *polyjam_workspace*: This is where the main part of the work happens. Here you can define problems and execute the generation of new solvers.
4) *polyjam_solvers*: This folder will receive the generated C++ solvers.

In order to use the library it first has to be build. The first thing to do is setting the correct console command to execute the Macaulay2 binary. Open the file *polyjam_generator/CMakeLists.txt* and go to line 27. It should contain the following:

```
IF(APPLE)
 SET(MACAULAYCOMMAND "'/./Applications/Macaulay2-1.7/bin/M2'")
ELSE()
 SET(MACAULAYCOMMAND "'M2'")
ENDIF()
```

"M2" should work under Ubuntu, but, depending on the version of Macaulay, the APPLE-command may have to be modified. After this is done, go to the main root folder and execute the following sequence of commands:

```
cd polyjam_generator
mkdir build
cd build && cmake .. && make
```

If everything goes well, this will complete the installation procedure. Note that of course a suitable C++ compiler needs to be installed on the system, as well as *cmake*.

Using the library

polyjam is a C++ library that generates C++ code. It may a bit counterintuitive at the beginning, but this means that we do not write script files to generate C++ code for our problems, but little C++ binaries that rely on the functionality of polyjam. The interaction with C++, however, is kept at a minimum. All you have to do is write a single file with a *main*-routine that defines the problem and terminates with a call to the code-generator. Generating the actual solver is then very easy as the library comes with a handy script that allows us to run the entire procedure (compilation and execution of the binary) through a single command line.

Let's assume that the problem you want to solve can be described by the acronym *mp1*. To create a new project, please create a new subfolder *mp1* inside the directory *polyjam_workspace*. Inside the new sub-folder, place a new empty file called *mp1.cpp*. It is important that this file has the same name than the subfolder. Open the file and enter the following:

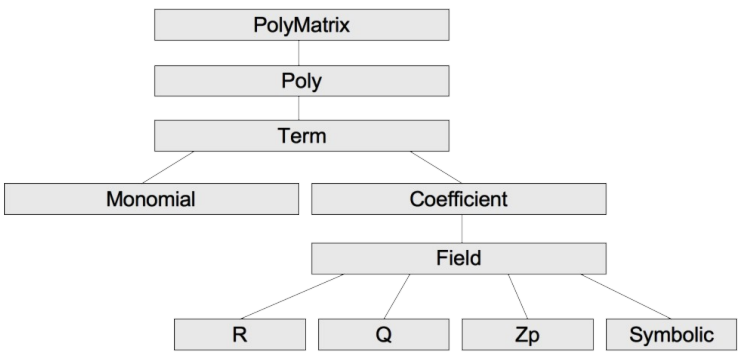
```
#include <polyjam/polyjam.hpp>

int main( int argc, char** argv )
{
 //initialize the random generator
 initGenerator();
}
```

We now have created the playground for generating a solver for problem *mp1*. The code generation can be triggered by opening the console, going to the directory *polyjam_workspace*, and entering

```
./generate mp1
```

This will run the entire procedure and create the solver (if possible). Note that until now our binary of course isn't doing anything. The rest of this documentation will explain how to actually define a problem, notably at the hand of two examples from the field of geometric computer vision.

The functionality of polyjam is easily explained by looking at the header structure inside *polyjam_generator/include/polyjam/*. The sub-folder *core* contains a hierarchy of elements that is required for composing polynomials. The ones that will be used the most are *Poly* and *PolyMatrix*, which define polynomials and polynomial matrices respectively. The sub-folder *fields* defines the various types of coefficients (i.e. field-members) that may be used. Although only *Symbolic* and *Zp* will be used in this tutorial, real numbers and rational numbers are available as well. The entire hierarchy looks as follows

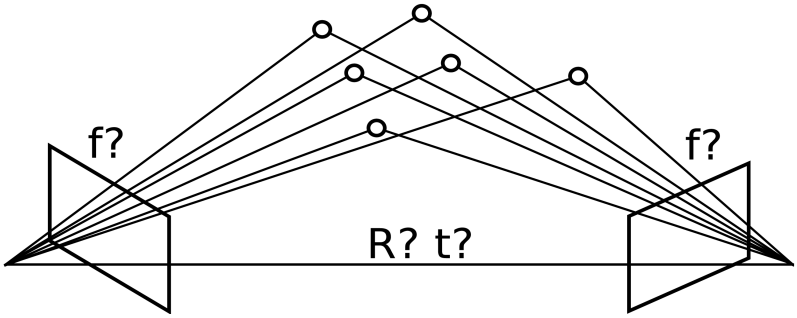


polyjam(.hpp) and *generator/methods* finally contain the machinery for generating the solver, *generator/CMatrix* defines the coefficient matrix where columns group the coefficients corresponding to a certain monomial, *generator/ExportMacaulay* is used for interaction with Macaulay2, and the sub-folder *math* contains a generic implementation of Gauss-Jordan elimination.

Note: initGenerator() makes sure that each time we run the binary a different random instance of a particular problem will be created. You can remove this line or replace it with alternative code if you intend to change this behaviour, and have repeatable results.

Example 1: Relative pose with unknown focal length

The first example we are looking at consists of finding the relative pose between two camera views given that the cameras are only calibrated up to an unknown focal length.



Let's denote the image measurements with $\boldsymbol{x}=\begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$ and $\boldsymbol{x}'=\begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix}$. We assume that the input data is centred about the principal point, which can be easily done through the often valid assumption that the principal point lies in the centre of the image. The unknown variables therefore are the relative translation $\boldsymbol{t}$, the relative rotation $\boldsymbol{R}$ between the two view-points, as well as the unknown focal length f.

As explained in Hartley & Zissermann 2004 (*Multiple View Geometry*), the frame-to-frame correspondences are constrained by the fundamental matrix $\boldsymbol{F}$ following the law

$$\boldsymbol{x}'^{T}\boldsymbol{F}\,\boldsymbol{x}=0 \quad .$$

$\boldsymbol{F}$ is a function of the relative transformation as well as the intrinsic camera parameters. By using the Direct Linear Transformation (DLT), we can reformulate the above expression such that the parameters of the fundamental matrix lie in the null space of a matrix which only depends on the input correspondences:

$$DLT(\boldsymbol{x},\boldsymbol{x}')\,vectorize(\boldsymbol{F})=0 \quad .$$

$vectorize(\boldsymbol{F})$ is a vector containing the 9 elements of $\boldsymbol{F}$. In the relative pose with unknown focal length problem, the number of unknowns is 6 (3 for rotation, 2 for the direction of the translation [the magnitude is arbitrary], and 1 for the focal length). In consequence, the problem must be solvable with 6 input correspondences.

If using 6 correspondences in the DLT transform, we obtain

$$\begin{bmatrix} DLT(\boldsymbol{x}_1, \boldsymbol{x}'_1) \\ DLT(\boldsymbol{x}_2, \boldsymbol{x}'_2) \\ \cdots \\ DLT(\boldsymbol{x}_6, \boldsymbol{x}'_6) \end{bmatrix} vecorize(\boldsymbol{F}) = 0 \quad .$$

The fundamental matrix therefore must lie in a nullspace spanned by the three right-most null-space vectors of

$$\begin{bmatrix} DLT(\boldsymbol{x}_1, \boldsymbol{x}'_1) \\ DLT(\boldsymbol{x}_2, \boldsymbol{x}'_2) \\ \cdots \\ DLT(\boldsymbol{x}_6, \boldsymbol{x}'_6) \end{bmatrix} .$$

Denoting the fundamental matrices corresponding to those three null-space vectors with $\boldsymbol{F}_0$, $\boldsymbol{F}_1$, and $\boldsymbol{F}_2$, the solution is given by

$$\boldsymbol{F} = \boldsymbol{F}_0 + \lambda_1 \boldsymbol{F}_1 + \lambda_2 \boldsymbol{F}_2 \quad .$$

Note that only two linear combination weights λ_1 and λ_2 are necessary as the overall scale of $\boldsymbol{F}$ is arbitrary.

We have used our measurements and the DLT transform to reduce the number of unknowns to 2, and we can now use the above form and additional constraints on the fundamental or—introducing the last unknown, namely the focal length—the essential matrix in order to fully solve the problem.

The first equation is very simple. In order for a fundamental matrix to be a correct fundamental matrix, it has to fulfill the condition that its determinant is zero. The first equation therefore is

$$det(\boldsymbol{F}) = 0 \quad ,$$

where $\boldsymbol{F} = \boldsymbol{F}_0 + \lambda_1 \boldsymbol{F}_1 + \lambda_2 \boldsymbol{F}_2$.

We now introduce the unknown focal length and the link between the fundamental matrix and the essential matrix:

$$\boldsymbol{K}^{-T} \boldsymbol{E} \boldsymbol{K}^{-1} = \boldsymbol{F} \rightarrow \boldsymbol{E} = \boldsymbol{K}^T \boldsymbol{F} \boldsymbol{K} \quad ,$$

where $\boldsymbol{K} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix}$.

The constraint on the essential matrix that we can exploit is given by

$$2\boldsymbol{E}\boldsymbol{E}^T\boldsymbol{E} - trace(\boldsymbol{E}\boldsymbol{E}^T)\boldsymbol{E} = \boldsymbol{0} \quad .$$

Following the algorithm as originally presented in Stewenius et al. CVPR'05 (*A minimal solution for relative pose with unknown focal length*), we replace the essential matrix with the above combination of the fundamental matrix and the focal length matrix to obtain the following constraint on our three unknowns:

$$2\,\boldsymbol{K}^T\boldsymbol{F}\boldsymbol{K}\boldsymbol{K}^T\boldsymbol{F}^T\boldsymbol{K}\,\boldsymbol{K}^T\boldsymbol{F}\boldsymbol{K}-trace\left(\boldsymbol{K}^T\boldsymbol{F}\boldsymbol{K}\boldsymbol{K}^T\boldsymbol{F}^T\boldsymbol{K}\right)\boldsymbol{K}^T\boldsymbol{F}\boldsymbol{K}=\boldsymbol{0}\quad .$$

Using the fact that $trace\left(\boldsymbol{K}^T\boldsymbol{F}\boldsymbol{K}\boldsymbol{K}^T\boldsymbol{F}^T\boldsymbol{K}\right)=trace\left(\boldsymbol{F}\boldsymbol{K}\boldsymbol{K}^T\boldsymbol{F}^T\boldsymbol{K}\boldsymbol{K}^T\right)$, we can transform this constraint into

$$\begin{gathered}\boldsymbol{K}^T\left[2\,\boldsymbol{F}\boldsymbol{K}\boldsymbol{K}^T\boldsymbol{F}^T\boldsymbol{K}\,\boldsymbol{K}^T\boldsymbol{F}-trace\left(\boldsymbol{F}\boldsymbol{K}\boldsymbol{K}^T\boldsymbol{F}^T\boldsymbol{K}\boldsymbol{K}^T\right)\boldsymbol{F}\right]\boldsymbol{K}=\boldsymbol{0}\\ \rightarrow 2\,\boldsymbol{F}\boldsymbol{K}\boldsymbol{K}^T\boldsymbol{F}^T\boldsymbol{K}\,\boldsymbol{K}^T\boldsymbol{F}-trace\left(\boldsymbol{F}\boldsymbol{K}\boldsymbol{K}^T\boldsymbol{F}^T\boldsymbol{K}\boldsymbol{K}^T\right)\boldsymbol{F}=\boldsymbol{0}\end{gathered}\quad .$$

By replacing $\boldsymbol{K}\boldsymbol{K}^T=w^{-1}\boldsymbol{Q}$, where $w=\frac{1}{f^2}$ and $\boldsymbol{Q}=\begin{bmatrix}1&0&0\\0&1&1\\0&0&w\end{bmatrix}$, and by omitting the factor w^{-1} , we finally obtain

$$2\,\boldsymbol{F}\boldsymbol{Q}\boldsymbol{F}^T\boldsymbol{Q}\boldsymbol{F}-trace\left(\boldsymbol{F}\boldsymbol{Q}\boldsymbol{F}^T\boldsymbol{Q}\right)\boldsymbol{F}=\boldsymbol{0}\quad .$$

Now that we have our constraints, let's see how we can use this to set up our problem in polyjam. The example is included in the library in the sub-folder *sw6pt*. The first bit of the code after the initialization looks as follows:

```
//initialize the input with random coefficients
PolyMatrix F1(Poly::zeroSZ(nu),3,3);
PolyMatrix F2(Poly::zeroSZ(nu),3,3);
PolyMatrix F3(Poly::zeroSZ(nu),3,3);

for( int r = 0; r < 3; r++ )
{
 for( int c = 0; c < 3; c++ )
 {
  //chose smart symbolic names that represent
  //the location of the measurement in the final code
  //(e.g. inside a vector or-in this case-a matrix)
  stringstream name1; name1 << "F1(" << r << "," << c << ")";
  stringstream name2; name2 << "F2(" << r << "," << c << ")";
  stringstream name3; name3 << "F3(" << r << "," << c << ")";

  F1(r,c) = Poly::SrandZ(name1.str(),nu);
  F2(r,c) = Poly::SrandZ(name2.str(),nu);
  F3(r,c) = Poly::SrandZ(name3.str(),nu);
 }
}
```

This part sets up the random problem in the finite (prime) field. The first three rows create three matrices of size 3x3. *Poly::zeroSZ(nu)* is a dummy zero variable that we add in order to tell the constructor the type of the matrix. *nu* is a variable that is used throughout the code to indicate the number of unknowns, by which we know how to construct the monomials for each term in each polynomial (even if these monomials initially equal to one for the input measurements, they will have to be initialised correctly). The cascaded for-loops then fill these matrices with random elements from the field Zp (the field of integer coefficients modulo some prime number p). However, this is not everything there is to the initialisation. As you may have noticed, the initialiser

for the matrices is of type *zeroSZ*. *SZ* here stands for *Symbolic* AND *Zp*. One strength of polyjam is that coefficients of terms can have a double representation, for instance one in a finite prime field, and one as a symbolic term. This makes perfect sense, as polyjam in fact needs those two representations for input measurements: First, it needs to have a finite random coefficient for each measurement variable in order to find the Groebner basis in a finite field. Second, it also needs to know the name of this variable through which it can be communicated in the final code. This is exactly why we have to fill the elements of our three input matrices with *Poly::SrandZ(namex.str(),nu)*, which creates a term with a double coefficient, one being a random member of Zp, and one being a symbol represented by *name*. We chose a smart name that allows us to group measurement inputs in a meaningful way. The name in fact indicates that the variable is a particular element of a matrix. Although this is optional, it is smart as it allows us to lateron communicate the measurements to our solver through 3 matrices rather than 27 individual variables. Note that, in contrary to other solver generators, this variable grouping is implicit, nothing more needs to be added. In particular, it is made possible as polyjam allows you to utilise any character—including brackets and any other special characters—to give variables a name. Matlab is different in this regard.

Ok, we are ready to move over to the second part of our code.

```
//initialize unknowns
Poly x = Poly::uSZ(1,nu);
Poly y = Poly::uSZ(2,nu);
Poly w = Poly::uSZ(3,nu);
```

Poly::uSZ simply represents a polynomial term with a coefficient again having a double representation (in a finite and a symbolic field), this time however with a monomial representing one of the unknowns. The integer *x* in the constructor will cause the construction of the *x*-th unknown. *x* and *y* represent the unknowns λ_1 and λ_2 in the theory part.

The next part then in fact generates the equations through which we constrain our problem

```
//prepare some intermediate variables
PolyMatrix F = F1 * x + F2 * y + F3;
PolyMatrix Ft = F.transpose();
PolyMatrix Q(Poly::oneSZ(nu),3,3,true);
Q(2,2) = w.clone();
PolyMatrix FQFtQ = F*Q*Ft*Q;
PolyMatrix te = FQFtQ * F * Poly::constSZ(2,nu) - F * FQFtQ.trace();
```

The first line parametrises $\boldsymbol{F}$ as a function of our three null-space vectors and λ_1 and λ_2, as explained in the theory. The second line simply creates a copy of this polynomial matrix that represents its transpose. In the third and fourth line, we then create matrix $\boldsymbol{Q}$. We first initialise it as a 3x3 matrix with a constant 1 (again with double field representation). Note the *true* as a fourth parameter though, which causes the initialisation to be along the diagonal only (all other elements are set to 0). So this in fact constructs the identity matrix. In the next line, we then replace then lower-right element by *w,* thus completing the initialisation of $\boldsymbol{Q}$. To conclude, the last two lines construct the 3x3 essential matrix constraint as a function of the fundamental matrix and the unknown focal length, again as explained in the theory part.

The arithmetics of polyjam allow to do the basic things needed for constructing and manipulating polynomials, but it is not perfect. Due to the class-hierarchy and absence of public operators, for instance, scalars have to be multiplied to matrices from the right side. Furthermore, associations are lazy, which has the advantage that the result of a large arithmetic expression does not need to be copied for the purpose of association. On the downside, if we simply want to copy a polynomial to a different location (for instance to a certain position in a matrix as in *Q(2,2) = w.clone()*), we need to add a *.clone()* call in order to make this a hard copy (it may not be important though in this particular case because *w* isn't used anywhere else). Future versions of polyjam will improve on this matter, such that these little issues do no longer need specific care.

We now have created all equations in the finite field, including the coefficients' symbolic dependency on the input measurements. The final task before calling the solver generator consists of adding all equations to a list of *Poly*-pointers.

```
//store all equations
list<Poly*> eqs;
eqs.push_back(new Poly(F.determinant()));
for( int r = 0; r < 3; r++ )
{
 for( int c = 0; c < 3; c++ )
   eqs.push_back(new Poly(te(r,c)));
}
```

The second line in this piece of code notably adds the determinant constraint on the fundamental matrix that was still missing up to this point.

We finish by calling the solver generator.

```
//****** Part 2: Generate the solver ***************
execGenerator( eqs, string("sw6pt"), string("Eigen::Matrix3d & F1,
Eigen::Matrix3d & F2, Eigen::Matrix3d & F3") );
```

The parameters are explained as follows:

1) *eqs* is the list of polynomial constraints on our three unknowns, with coefficients having a representation in both the finite prime field, as well as a symbolic one that explains their construction as a function of the original input measurements.
2) *string("sw6pt")* is a name we give to our solver.
3) *string("Eigen::Matrix3d & F1, Eigen::Matrix3d & F2, Eigen::Matrix3d & F3")* finally represents the parameter list through which we can communicate the measurements to the generated solver. Note that the type and name of these parameters needs to agree with the smart "indexing" symbolic names we gave to our measurements in the first part of the code.

This in fact concludes the work that has to be done. In order to trigger the generation of the code, simply go to the folder *polyjam_workspace*, and execute the command

```
./generate sw6pt
```

The progress of the code generation can be observed by following the console output. It should look as follows:

```
Laurents-MBP:polyjam_workspace laurent$ ./generate sw6pt
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/laurent/ldevel/polyjam/polyjam_workspace/build
Scanning dependencies of target sw6pt
[100%] Building CXX object CMakeFiles/sw6pt.dir/sw6pt/sw6pt.o
Linking CXX executable ../bin/sw6pt
[100%] Built target sw6pt

polyjam
Copyright (C) 2015 Laurent Kneip, The Australian National University
This program comes with ABSOLUTELY NO WARRANTY; It is free software:
you can redistribute it and/or modify it under the terms of the GNU General Public License

Analysing the Groebner basis in Macaulay2 ...
The basis monomials are:
| 1 x_1 x_1^2 x_1x_2 x_1x_2x_3 x_1x_3 x_1x_3^2 x_2 x_2^2 x_2^2x_3 x_2x_3 x_2x_3^2 x_3 x_3^2 x_3^3 |

Finding the degree of expansion.
The leading Monomials that we are interested in are:
x_1*x_2*x_3^2
x_2^2*x_3^2
x_1*x_3^3
x_2*x_3^3
x_3^4
x_1^2*x_3
Trying out degree 2
Template size: 66x88
Did not find all monomials.
Trying out degree 3
Template size: 107x130
Did not find all monomials.
Trying out degree 4
Template size: 158x182
Found all monomials.
Generating the super-linear expanders.
Starting the solver generation.
Pre-elimination is done.
Extracted the polynomials that are needed for composing the Action matrix.
Trying to remove equations that are unnecessary.
350 .. 349 .. 347 .. 343 .. 335 .. 319 .. 319 .. 303 .. 303 .. 303 .. 295 .. 295 .. 295 .. 295 ..
293 .. 293 .. 293 .. 292 .. 292 .. 292 .. 291 .. 289 .. 285 .. 277 .. 261 .. 229 .. 229 .. 229 ..
213 .. 213 .. 213 .. 205 .. 205 .. 205 .. 201 .. 201 .. 201 .. 201 .. 200 .. 200 .. 200 .. 200 ..
200 .. 200 .. 200 .. 200 .. 200 .. 200 .. 199 .. 197 .. 193 .. 185 .. 169 .. 169 .. 169 .. 169 ..
165 .. 165 .. 165 .. 163 .. 163 .. 163 .. 163 .. 162 .. 162 .. 162 .. 161 .. 159 .. 155 .. 147 ..
131 .. 131 .. 115 .. 115 .. 115 .. 115 .. 111 .. 111 .. 111 .. 111 .. 111 .. 111 .. 111 .. 111 ..
111 .. 111 .. 111 .. 111 .. 111 .. 110 .. 108 .. 104 .. 96 .. 96 .. 96 .. 92 .. 92 .. 92 .. 90 .. 90
.. 90 .. 89 .. 89 .. 89 .. 89 .. 88 .. 86 .. 86 .. 86 .. 86 .. 85 .. 85 .. 85 .. 84 .. 82 .. 78 ..
70 .. 70 .. 70 .. 66 .. 66 .. 66 .. 64 .. 64 .. 64 .. 64 .. 64 .. 64 .. 64 .. 64 .. 64 ..
64 .. 64 .. 63 .. 61 .. 57 .. 57 .. 57 .. 57 .. 56 .. 56 .. 56 .. 56 .. 56 .. 55 .. 55 .. 55 ..
55 .. 54 .. 54 .. 54 .. 54 .. 54 .. 54 .. 54 .. 53 .. 53 .. 53 .. 53 .. 53 .. 53 .. 53 .. 53 ..
53 .. 53 .. 53 ..
I am done with this round. Original height of template was 350. Now it is 53.
Removing unused monomials.
Final template size: 53x72
Reordering the monomials.
Extracting the code
```

The first six lines automatically configure the configuration of a project to build the binary, and

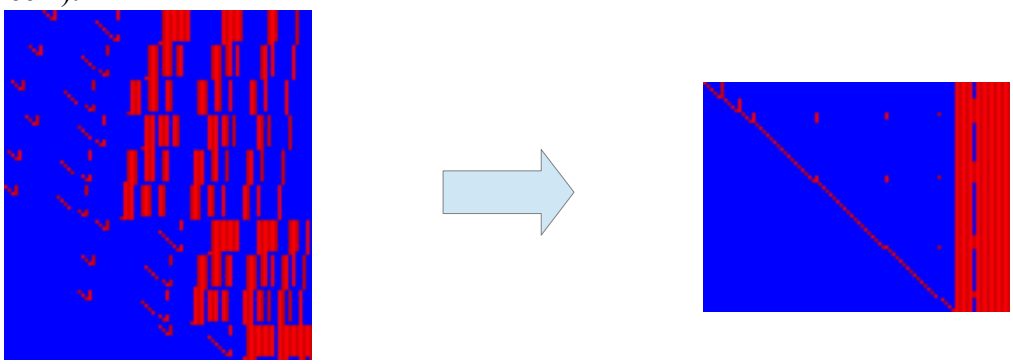
launch compilation. We then can see four blocks of red, blue, red, and again blue background colour. These outputs notably correspond exactly to the four subsequent steps that polyjam takes care of after the user has constructed the problem (external call to Macaulay2, search for the expander degree, removal of redundant equations, and code generation). The generated solver can then be found in the *polyjam_solvers/sw6pt* folder.

As mentioned in the installation instructions, polyjam also depends on OpenCV to visualise the elimination templates, notably along the solver generation process. This visualisation can be activated by appending a *true* as a fourth parameter to the final solver generation call, as follows

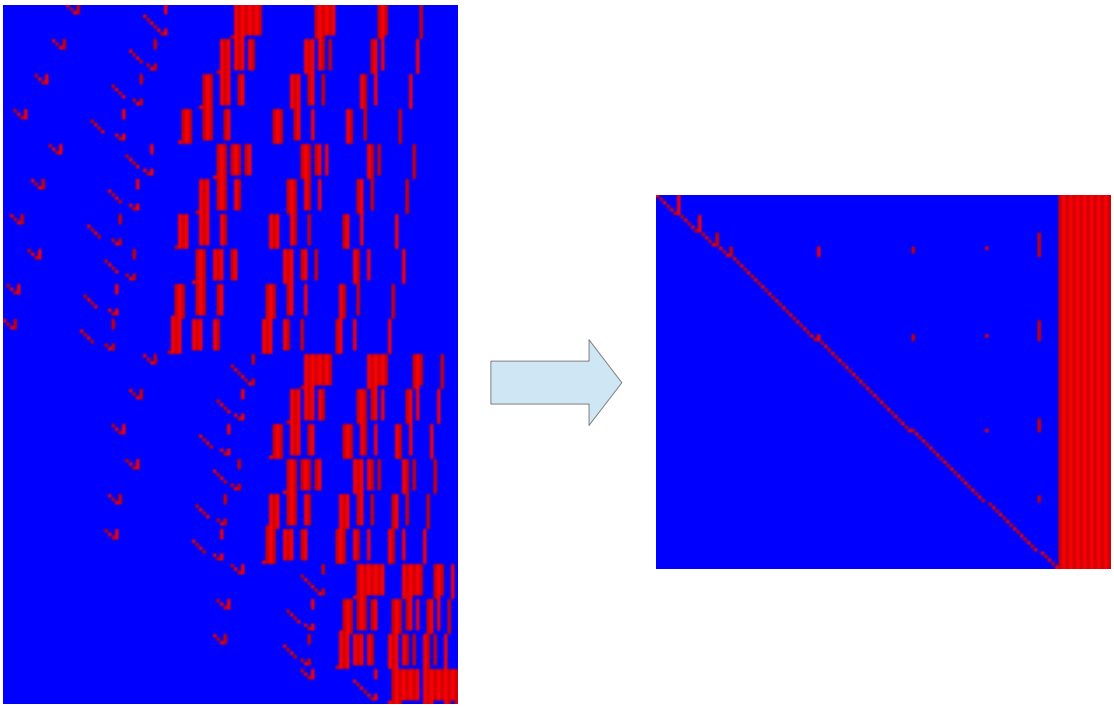
```
//****** Part 2: Generate the solver ***************
execGenerator( eqs, string("sw6pt"), string("Eigen::Matrix3d & F1,
Eigen::Matrix3d & F2, Eigen::Matrix3d & F3"), true );
```

Regenerating the code will then go through a sequence of images, which can be stepped by highlighting the image and each time pressing the escape key. The first *2n* images represent the templates occurring in step 3 of the solver generation process, where we iteratively add all equations up to a certain, increasing degree until we have found all equations (the degrees notably are *2, ..., n+1*). The number of images is *2n* since we each time see the template before and after the Gauss-Jordan elimination. In our case, *n=3* and the sequence looks as follows:

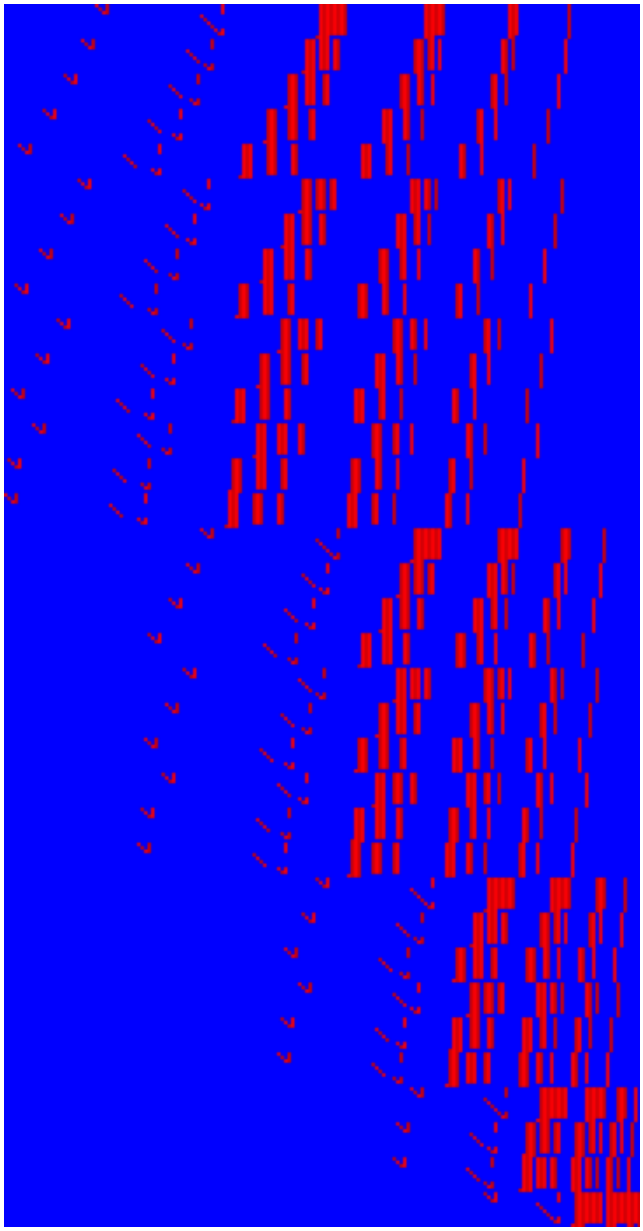
n = 1 (degree 2):

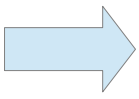


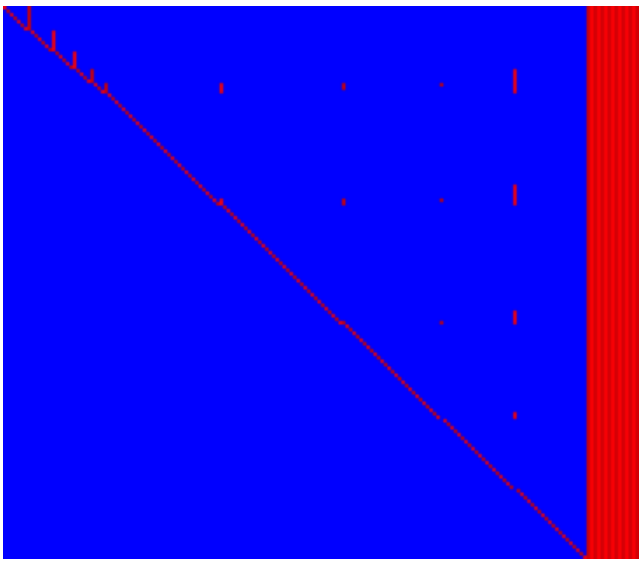
n = 2 (degree 3):



$n = 3$ (degree 4):

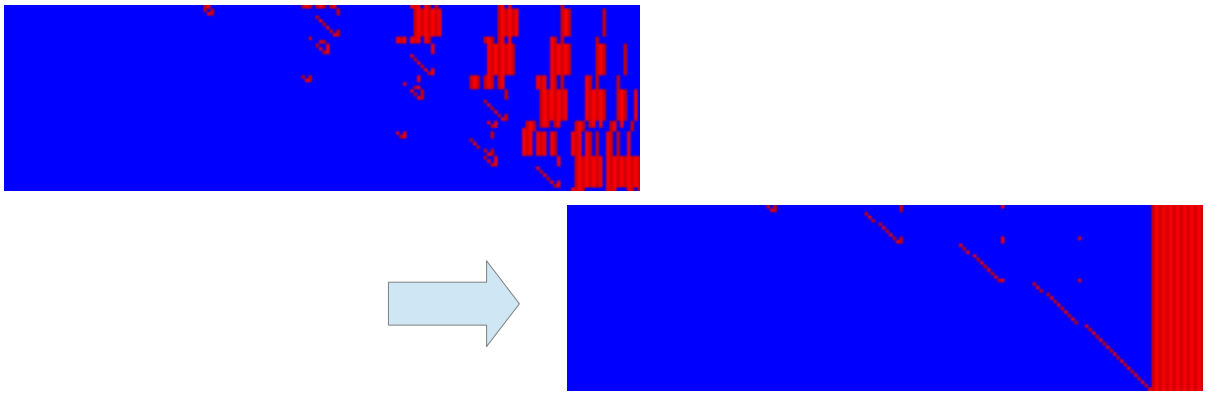




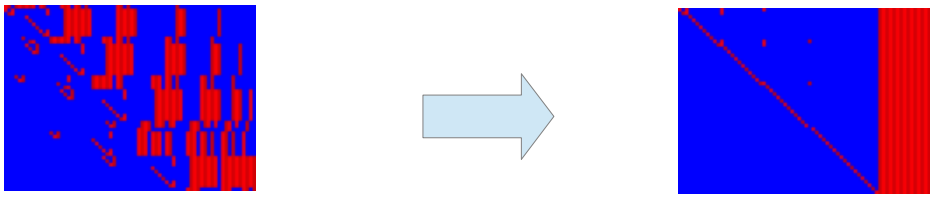


After this we have found all the equations, and are ready to move on to step 4, which consists of eliminating the unnecessarily added equations. At last, we can see a sequence of 3 pairs of images:

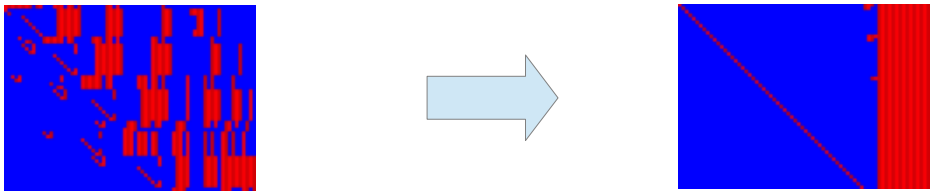
The resulting elimination template without any changes:



The resulting elimination template with the unnecessary monomials removed (=empty columns removed):



The final elimination template where the monomials are reordered:



The latter form has the advantage that elimination now results in a diagonal block matrix on the left, allowing us to use efficient matrix inversion techniques to perform the elimination.

Congratulations if you made it until here, you just went through the entire core functionality of the library.

Playing around with polyjam

This is a quick summary of further features that may be useful:

1) Polynomials may simply be printed into the console by executing *print()*. They may also be converted to a string using *getString()*. Note that *getString()* takes a boolean parameter to activate C-style strings where powers of variables are constructed using the *pow()* function. If setting this variable to *false*, powers will be represented using ^, the result of which can then be interpreted by Macaulay2.
2) The library allows for automatic degree limitation of polynomials. This is a somewhat unexplored feature, but for certain complicated problems lower degree approximations about zero may indeed lead to valuable solutions, as already demonstrated in the geometric vision community, where linear approximations to the rotation matrix have been explored

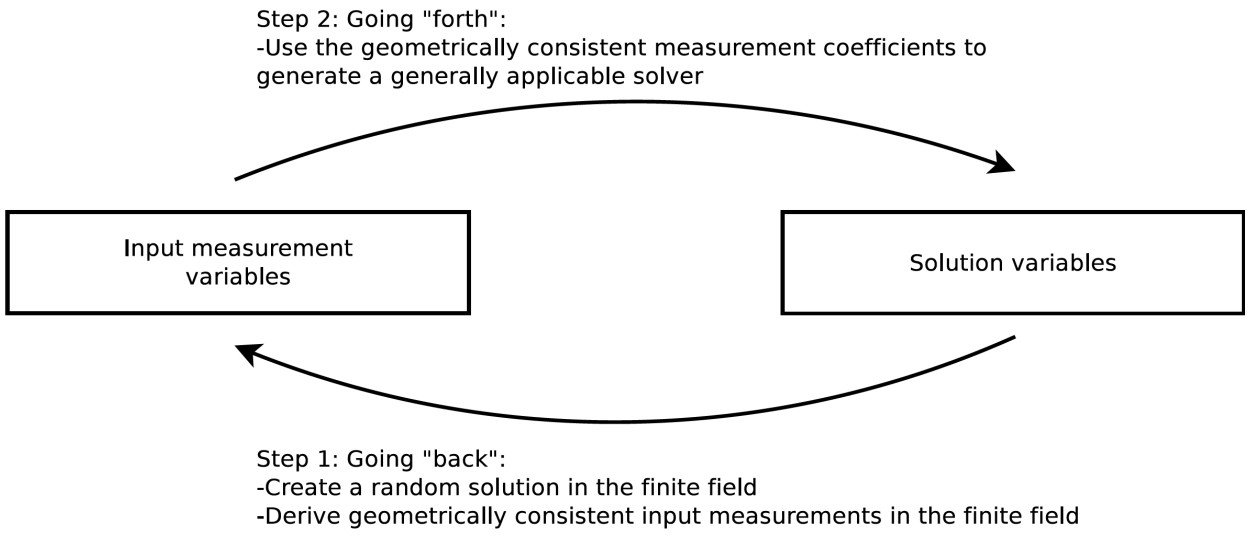
for pose estimation problems.

3) The library by default uses the graded-reverse-lexicographical monomial ordering, but different orderings such as lexicographical, and reverse-lexicographical are available as well. This will have to be indicated explicitly in the polynomial constructors.
4) The Macaulay scripts that are used to compute the Groebner basis in step 2 can be found in the folder *polyjam_workspace/sw6pt/M2script*.
5) The library has successfully been used to create solvers where symmetry in the solution space is taken into account (this leads to a reduction of the size of the elimination template and the number of solutions). Please contact the author if you are interested in using this functionality. The idea has been presented in Ask et al. ICPR'12 (*Exploiting p-fold symmetries for faster polynomial equation solving*).

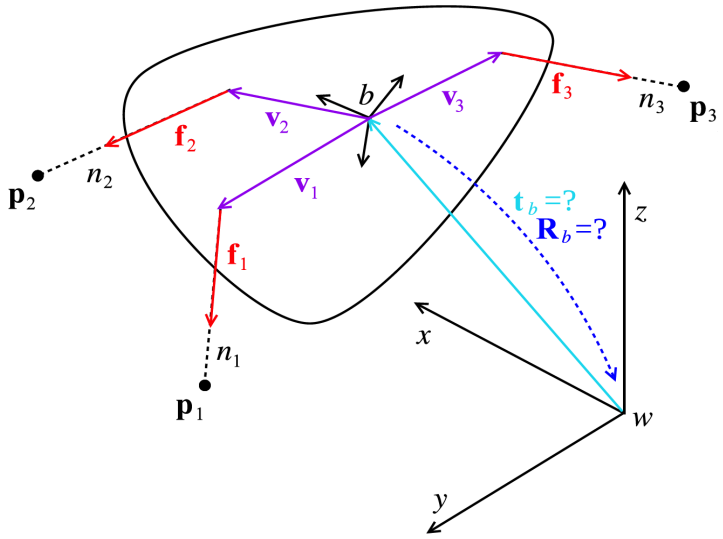
Example 2: Absolute pose of a generalised camera

Before we conclude this brief tour, let us take a look at a final example to explain another important feature of polyjam. You may have noticed that in Example 1 we simply constructed three random fundamental matrices to span the null-space of our solution, however without explicitly adding the constraint that these vectors should be orthogonal to each other. For this particular problem it may not matter, but it is important to realise that the measurements in a typical problem are not independent of each other. More specifically, systematic zero cancellations along the elimination processes may or may not happen depending on whether or not these dependencies are correctly reflected in the measurements, even in the finite field! The following is about **the importance of creating geometrically consistent problems even in the finite field**, and how this can be easily done using polyjam.

We introduce a “back-and-forth” strategy to do so. Frequently, the type of variables in a geometric problem that are actually fully independent include the ones that we want to solve for. An example is given by the problem of finding the absolute pose (translation and orientation) of a camera. If not using the minimal number of points, the measurements are no longer fully independent. In the ideal case, taking all the world points but only three of the image point measurements allows us to derive all the other image point measurements as well, which clearly proves a dependency in the measurements. Truly independent variable types are given by the world points and the parameters of the camera pose. The correct way of setting up a geometrically consistent pose estimation problem in the finite field therefore consists of creating a random solution in terms of the final pose, deriving geometrically consistent input measurements from there (going “back”), and then constructing the equations and generating the solver from here (going “forth”). This notably forms the “back-and-forth” concept:



The particular problem we are looking at is the generalised absolute pose problem. Let us first define it.



The generalised absolute pose problem consists of finding the pose of a body frame (rotation $\boldsymbol{R}$ and translation $\boldsymbol{t}$) from which we observe three world points $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, and $\boldsymbol{p}_3$. The particularity with respect to the "normal" absolute pose problem is given by the fact that the measurement rays do no longer intersect in a common point (called the focal point for a regular camera), but they have arbitrary moments with respect to the body centre. A practically relevant example which would deliver such measurements would be a calibrated multi-camera system with three cameras, where we then would compute the pose of the entire system using one a single 2D-3D correspondence from each one of the cameras. Our measurement input is given by the world points, vectors $\boldsymbol{f}_1$, $\boldsymbol{f}_2$, and $\boldsymbol{f}_3$ denoting the direction of the rays, and vectors $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, and $\boldsymbol{v}_3$ denoting the coordinates of one point on each ray expressed inside the body frame.

We will apply the "back-and-forth" concept, leading to two major parts in the code. The first one consists of generating geometrically consistent coefficients in the finite field. We start by generating a random transformation, random world points, and random moments. These are independent variables, and deriving the ray directions then relies on a straightforward application of the generalised camera's projection equation

$$\boldsymbol{f}_i = \boldsymbol{R}^T\left(\boldsymbol{p}_i - \boldsymbol{t}\right) - \boldsymbol{v}_i \quad .$$

Note that the scale of the resulting direction vector is irrelevant, it can be arbitrarily rescaled. Also note that the application of the "back-and-forth" strategy may not be required in the present example, we could just as well create random direction vectors directly, there would always remain at least one camera pose for which the points and image plane measurements would make perfect sense. However, we start here from the pose parameters for the purpose of demonstrating the idea. The second step then consists of coming up with constraints on our pose that allow us to solve the problem. We use here the parameterisation introduced in Kneip et al. 2013 (*Using Multi-Camera Systems in Robotics*). It relies on the unknown depths n_i as helping variables. We have

$$\boldsymbol{R}\left(\boldsymbol{f}_i n_i + \boldsymbol{v}_i\right) + \boldsymbol{t} = \boldsymbol{p}_i \quad .$$

By combining pairwise equations, we can easily eliminate the unknown translation from the equations, and obtain

$$\boldsymbol{f}_i n_i - \boldsymbol{f}_j n_j + \boldsymbol{v}_i - \boldsymbol{v}_j = \boldsymbol{R}^T\left(\boldsymbol{p}_i - \boldsymbol{p}_j\right) \quad ,$$

where *(i,j)* equals *(1,2)*, *(2,3)*, or *(3,1)*.

Ok, we are ready to look at this in the code. Please open *polyjam_workspace/gen3pt/gen3pt.cpp*. The first block looks as follows

```
//initialize the random generator
initGenerator();
size_t nu = 6; //the number of unknowns in the problem
size_t numberBearingVectors = 3;
```

As usual, this initialises the random generator and sets the handy variable *nu* which indicates the number of unknowns. The code also sets *numberBearingVectors*, so the entire first part of the code can be reused for a problem with *n* points as well by simply changing this variable. The next bit of code then corresponds to the instantiation of a random problem through independent variables (in this case the pose, the camera configuration, and the world points).

```
//****** Part 1: get independent random variables *******//

//random rotation
std::cout << "generating random rotation" << std::endl;
PolyMatrix cay_gt(Poly::zeroZ(nu),3,1);
for( int i = 0; i < 3; i++ )
  cay_gt[i] = Poly::randZ(nu);

Poly scale2 = Poly::oneZ(nu) + (cay_gt[0]*cay_gt[0]) + (cay_gt[1]*cay_gt[1]) +
(cay_gt[2]*cay_gt[2]);
Poly scale2_inv = Poly::oneZ(nu).leadingTerm() / scale2.leadingTerm();

PolyMatrix R_gt(Poly::zeroZ(nu),3,3);
R_gt(0,0) = scale2_inv * ( Poly::oneZ(nu) + (cay_gt[0]*cay_gt[0]) -
(cay_gt[1]*cay_gt[1]) - (cay_gt[2]*cay_gt[2]) );
R_gt(1,1) = scale2_inv * ( Poly::oneZ(nu) - (cay_gt[0]*cay_gt[0]) +
(cay_gt[1]*cay_gt[1]) - (cay_gt[2]*cay_gt[2]) );
R_gt(2,2) = scale2_inv * ( Poly::oneZ(nu) - (cay_gt[0]*cay_gt[0]) -
(cay_gt[1]*cay_gt[1]) + (cay_gt[2]*cay_gt[2]) );
R_gt(0,1) = scale2_inv * ( Poly::constZ(2,nu) * (cay_gt[0]*cay_gt[1]-cay_gt[2]) );
R_gt(0,2) = scale2_inv * ( Poly::oneZ(nu) * (cay_gt[0]*cay_gt[2]+cay_gt[1]) );
R_gt(1,2) = scale2_inv * ( Poly::oneZ(nu) * (cay_gt[1]*cay_gt[2]-cay_gt[0]) );
R_gt(1,0) = scale2_inv * ( Poly::oneZ(nu) * (cay_gt[0]*cay_gt[1]+cay_gt[2]) );
R_gt(2,0) = scale2_inv * ( Poly::oneZ(nu) * (cay_gt[0]*cay_gt[2]-cay_gt[1]) );
R_gt(2,1) = scale2_inv * ( Poly::oneZ(nu) * (cay_gt[1]*cay_gt[2]+cay_gt[0]) );

//random translation
std::cout << "generating random translation" << std::endl;
PolyMatrix t_gt(Poly::zeroZ(nu),3,1);
for( int i = 0; i < 3; i++ )
  t_gt[i] = Poly::randZ(nu);

//random world points and offsets
std::cout << "generating random world points and offsets" << std::endl;
std::vector<PolyMatrix> wps;
std::vector<PolyMatrix> vs;
for( int i = 0; i < (int) numberBearingVectors; i++ )
{
  wps.push_back(PolyMatrix(Poly::zeroZ(nu),3,1));
  vs.push_back(PolyMatrix(Poly::zeroZ(nu),3,1));

  for( int j = 0; j < 3; j++ )
```

```
    {
     wps.back()[j] = Poly::randZ(nu);
     vs.back()[j] = Poly::randZ(nu);
    }
  }
```

Note that for "going back", we do not need a symbolic representation. It is only used to derive geometrically consistent coefficients in Zp. In consequence, polynomials are created using the constructors *zeroZ*, *oneZ*, *constZ*, and *randZ*, rather than *zeroSZ*, *oneSZ*, *constSZ*, and *randSZ*. For the rest, the code is pretty much self-explanatory. The first sub-block consists of creating a 3-vector of random Cayley-coefficients. The second and third sub-block consist of transforming the Cayley parameters into a rotation matrix:

https://en.wikipedia.org/wiki/Rotation_matrix#Skew_parameters_via_Cayley.27s_formula

The next sub-block then consists of constructing a random 3-vector for the translation. The final sub-block consists of filling two standard vectors with random 3-vectors for the world points and the moments of the measurement rays in the generalised camera. The variables are called as follows:

1) Rotation: *R_gt*
2) Translation: *t_gt*
3) world points: *wps*
4) moments: *vs*

The only line that deserves more attention is

```
Poly scale12 = Poly::oneZ(nu) + (cay_gt[0]*cay_gt[0]) + (cay_gt[1]*cay_gt[1]) +
(cay_gt[2]*cay_gt[2]);
Poly scale12_inv = Poly::oneZ(nu).leadingTerm() / scale12.leadingTerm();
```

We here create the inverse of the scale that is necessary to create the rotation matrix. Unfortunately, the division is not yet implemented for polynomials, which is why we need to create the inverse beforehand by catching the leading terms of *oneZ(nu)* and *scale12*.

We are ready to move to the next block, which completes the idea of "going back" (i.e. from the solution to the input measurements).

```
//**** Part 2: extract the measurements with both Zp and symbolic representation ****//

std::cout << "extracting the image measurements" << std::endl;
std::vector<PolyMatrix> fs;

for( int i = 0; i < (int) numberBearingVectors; i++ )
{
  PolyMatrix f = R_gt.transpose() * ( wps[i] - t_gt ) - vs[i];

 //do some normalization here (just divide by the last coordinate!)
  Poly f_scale = f[2].clone();
  for( int j = 0; j < 3; j++ )
    f[j] = f[j].leadingTerm()/f_scale.leadingTerm();
```

```
 fs.push_back(PolyMatrix(Poly::zeroSZ(nu),3,1));

 for( int j = 0; j < 3; j++ )
 {
  std::stringstream name; name << "fs[" << i << "][" << j << "]";
  fs.back()[j] = Poly(Term(
   Coefficient(name.str()),
   f[j].leadingTerm().coefficient().clone(),
   Monomial(nu) ));
 }
}
```

We construct a standard vector *fs* which will receive 3-vectors representing the ray direction vectors. We first initialise individual direction vectors in the variable *f*. We normalise these vectors by dividing each element by the third coordinate. We then convert the coefficients to a double representation with a symbolic name next to the finite value. Towards this goal, we initialise a new 3-vector of *zeroSZ*-elements in *fs* which will receive the copy with double representation. The coordinates are copied one-by-one in a for-loop by initialising new polynomials with a single term that is constructed with both a string containing the symbolic name and a clone of the corresponding Zp-coefficient. *Monomial(nu)* will construct a new monomial with the right dimensionality but all exponents set to 0.

Before we can move over to the "forth"-part (extracting the final constraints based on which we generate the solver), we still have to do one thing. The independent original variables for the world points and the ray moments also need a symbolic name such that the generator can create proper code.

```
//****** Part 3: also lift the 3D points to the double representation ******//

std::cout << "lifting the 3D points" << std::endl;
std::vector<PolyMatrix> wps_double;
std::vector<PolyMatrix> vs_double;
for( int i = 0; i < (int) numberBearingVectors; i++ )
{
 wps_double.push_back(PolyMatrix(Poly::zeroSZ(nu),3,1));
 vs_double.push_back(PolyMatrix(Poly::zeroSZ(nu),3,1));

 for( int j = 0; j < 3; j++ )
 {
  std::stringstream name1; name1 << "wps[" << i << "][" << j << "]";
  wps_double.back()[j] = Poly(Term(
   Coefficient(name1.str()),
   wps[i][j].leadingTerm().coefficient().clone(),
   Monomial(nu) ));

  std::stringstream name2; name2 << "vs[" << i << "][" << j << "]";
  vs_double.back()[j] = Poly(Term(
   Coefficient(name2.str()),
   vs[i][j].leadingTerm().coefficient().clone(),
   Monomial(nu) ));
 }
}
```

This part of code does the same than what has been done before for the ray direction vectors. It constructs new copies of all the elements in *wps* and *vs*, adds a symbolic name to them, and stores them in new standard vectors called *wps_double* and *vs_double*. Note that in this example we

always use the smart, indexing variable names in the form *container[i][j]*. This is because we will communicate the measurement inputs through standard vectors containing Eigen::Vector3d matrices.

Having derived *fs*, *wps_double*, and *vs_double*, we now have constructed a geometrically consistent problem in the finite field, and are ready to move on to the "forth"-part (extracting the constraints and generating the actual solver).

```
//****** Part 4: get all the equations ****************//

std::cout << "Extracting the equations" << std::endl;

std::cout << "defining unknowns" << std::endl;
Poly x = Poly::uSZ(4,nu);
Poly y = Poly::uSZ(5,nu);
Poly z = Poly::uSZ(6,nu);

Poly n1 = Poly::uSZ(1,nu);
Poly n2 = Poly::uSZ(2,nu);
Poly n3 = Poly::uSZ(3,nu);

std::cout << "defining rotation" << std::endl;
PolyMatrix R(Poly::zeroSZ(nu),3,3);
R(0,0) = Poly::oneSZ(nu) + (x*x) - (y*y) - (z*z);
R(1,1) = Poly::oneSZ(nu) - (x*x) + (y*y) - (z*z);
R(2,2) = Poly::oneSZ(nu) - (x*x) - (y*y) + (z*z);
R(0,1) = Poly::constSZ(2,nu) * (x*y-z);
R(0,2) = Poly::constSZ(2,nu) * (x*z+y);
R(1,2) = Poly::constSZ(2,nu) * (y*z-x);
R(1,0) = Poly::constSZ(2,nu) * (x*y+z);
R(2,0) = Poly::constSZ(2,nu) * (x*z-y);
R(2,1) = Poly::constSZ(2,nu) * (y*z+x);

std::cout << "defining scale" << std::endl;
Poly scale = Poly::oneSZ(nu) + (x*x) + (y*y) + (z*z);

std::cout << "defining the M matrices" << std::endl;
PolyMatrix M1 = ( fs[0]*n1 - fs[1]*n2 + vs_double[0] - vs_double[1] ) * scale -
   R.transpose()*wps_double[0] + R.transpose()*wps_double[1];
PolyMatrix M2 = ( fs[1]*n2 - fs[2]*n3 + vs_double[1] - vs_double[2] ) * scale -
   R.transpose()*wps_double[1] + R.transpose()*wps_double[2];
PolyMatrix M3 = ( fs[2]*n3 - fs[0]*n1 + vs_double[2] - vs_double[0] ) * scale -
   R.transpose()*wps_double[2] + R.transpose()*wps_double[0];

std::cout << "extracting the equations" << std::endl;
std::list<Poly*> eqs;
for( size_t i = 0; i < 3; i++ )
  eqs.push_back(new Poly(M1[i]));
for( size_t i = 0; i < 3; i++ )
  eqs.push_back(new Poly(M2[i]));
for( size_t i = 0; i < 3; i++ )
  eqs.push_back(new Poly(M3[i]));

...
...
```

We use 3 Cayley-parameters *x*, *y*, and *z* to construct the rotation, and multiply the equations through with the scale factor appearing in the Cayley-to-rotation matrix conversion. The final call to the

solver generator is omitted here, as it is very long. Simply note that the variables are passed over as references to standard vectors containing *Eigen::Vector3d* elements.

The solver is finally generated by going to the *polyjam_workspace* folder, and executing the command

```
./generate gen3pt
```

The present example outlines the true potential of polyjam. It separates known and unknown variables since the very beginning, which enables a bottom-up calculation of geometrically consistent coefficients if working in a finite prime field. At the same time, the double representation of coefficients allows to still get the full algebraic expression needed for creating the coefficients.

Alternative solver generators simply work in the symbolic domain, and setting up problems with geometrically consistent coefficients in the final constraints is a very tedious task as it requires to recover every single instance of a certain variable from the complicated final expressions to replace them with the same random finite coefficient. In the Matlab solver generator of Kukelova et al., for instance, geometrically consistent behaviour can be activated by setting the option *cfg.InstanceGenerator* to *@gbs_RandomInstanceZp*. This will make the generator extremely slow, and consistency in the input measurements is still not necessarily guaranteed, as those simply remain random variables (i.e. it remains somewhat unclear how to implement the "back-and-forth" strategy in this solver, users may ultimately have to write their own instance generator function and dig into the arithmetics of finite fields). Polyjam works by default in a consistent way, and does so extremely efficiently. It also makes it easy to implement the "back-and-forth" strategy, as demonstrated in this example.

Testing a solver

You may wonder how to actually test one of the generated solvers. Looking into the output folder *polyjam_solvers/gen3pt*, you can find that besides the generated solver files *gen3pt.cpp* and *gen3pt.hpp*, there are already a number of files:

- *test.cpp*: This file generates random problems in the field of real numbers, and calls the solver multiple times in a loop to also measure the computational efficiency. The file allows to configure noise (and outliers), and transforms the final solution from Cayley parameters back to a real rotation. It also recovers the translation which was eliminated from the solutions process. The final result is printed into the console.
- *GaussJordan.hpp/.cpp*: An implementation of Gauss-Jordan for floating point numbers. This function is extremely important as it is called inside the generated solver for performing a pre-elimination on the original constraints (before filling the elimination template). This file will have to be copied over to compile any of the generated solvers.
- *experiment_helpers.hpp/.cpp*: For creating the random camera system and the random 2D-3D correspondences.
- *random_generators.hpp/.cpp*: For creating random points, rotations, and translations.
- *time_measurement.hpp/.cpp*: For measuring time.
- *CMakeLists.txt & modules*: For compiling the test example (it uses Eigen).

In order to compile the test, you have to go to the folder *polyjam_solver/gen3pt* and execute the commands

```
mkdir build
cd build && cmake .. && make && cd ..
```

The example can afterwards be executed with the command

```
cd bin
./test_gen3pt
```

This should print the correct solution and execution time into the console. Note that the test-files are mostly copied from the test-folder of the OpenGV library (https://github.com/laurentkneip/opengv), which contains many solvers that have been generated by polyjam. Further resources from the test-folder may be useful.

Contact

For further questions and use in proprietary applications, please contact the author of the library kneip.laurent@gmail.com

References

Cox, D.A., Little, J., O'Shea, D.: Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra, 3/e (Undergraduate Texts in Mathematics). Springer-Verlag New York, Inc., Secaucus, NJ, USA (2007).

Kukelova, Z., Bujnak, M., Pajdla, T.: Automatic generator of minimal problem solvers. In: Proceedings of the European Conference on Computer Vision (ECCV) (2008).

Hartley, R., Zisserman, A.: Multiple View Geometry in Computer Vision . Cambridge University Press, New York, NY, USA (2003).

Stewénius, H., Nistér, D., Kahl, F., Schaffalitzky, F.: A Minimal Solution for Relative Pose with Unknown Focal Length. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2005).

Ask, E., Yubin, K., Astrom, K.: Exploiting p-fold symmetries for faster polynomial equation solving. In: Proceedings of the International Conference on Pattern Recognition (ICPR) (2012).

Kneip, L., Furgale, P., Siegwart, R.: Using Multi-Camera Systems in Robotics: Efficient Solutions to the NPnP Problem. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA) (2013).