FreeCAD Sketcher Solver Architecture

Abdullah Tahiri

December 2018

To the FreeCAD Community

To Werner for reviewing tons of lines of code per year

Preface

Every year there are a couple of inquiries about how to get introduced into the solver code base. Motivated people with great ideas but limited time generally end up succumbing to the rather big entry energy required to understand how it is implemented.

It is not difficult for me to understand them. I have been myself overwhelmed by it. I am still. I have been though lucky enough of having several ingredients that have allowed me to reach a certain degree of understanding of what is going on: a) motivation to undertake changes, b) support from people with a better geometry knowledge and better mathematical background, c) time.

This document attempts to lower the entry point towards the code base of the solver.

It is my belief that only through a diversity of opinions and ideas we can actually collaborate into making significant and meaningful improvements.

This document should serve to gain a general common understanding allowing to identify improvements and enabling a discussion. For the rest, you are not alone. Make an account in the FreeCAD Community forum if you don't have already one. It is there that true collaboration starts.

Contents

Chapter 1

Introduction

This document is an introduction to solver architecture behind FreeCAD. It describes mainly the architecture and implementation of the solver. Other parts of the Sketcher are discussed only insofar as to enable to understand the former.

In order to save time to most readers, let's first describe what this is not. This document is not about how to use the Sketcher to draw a shape. This document is not about how to efficiently use the Sketcher. This document is not about general Sketcher programming.

This document tries to throw some light into how the solver actually works, and other than out of mere curiosity, it may be of little to no interest for $99.\overline{9}\%$ of the people. If you are still reading maybe you belong to that minority who is trying to:

[a)]

1. Introduce support for a new geometry in the Sketcher.

2. Introduce support for a new constraint in the Sketcher.

3. Try to solve a bug in the solver.

4. Get more information from the solver process to the user.

5. Introduce a new algorithm to the solver.

6. Improve an algorithm of the solver.

You will certainly get more information than you actually need for the first two items and you will still need much more information from elsewhere. You may want to search for the git commits when the support for Hyperbola and Parabola was introduced for more information.

For the next two items it should give you a fair overview of how things work. But it is probably for the last two items where this information may be specially valuable.

Chapter 2

Sketcher Architecture

2.1 Sketcher, solver and interface

A sketcher object, class *SketchObject*, contains all the geometry and constraints that define a given Sketch, as well as all the high level functions that can be applied to a sketch to add, delete or otherwise modify this geometry.

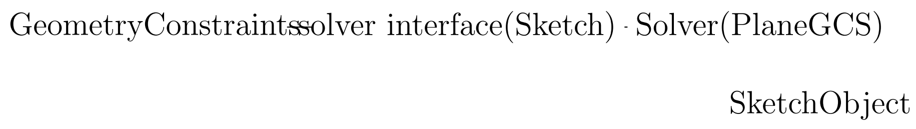
However, the sketcher object relies on specialised code, a geometric constraint system called PlaneGCS, in order to perform the actual solving of the constraints, which causes the geometry to adapt to those constraints.

The most important concept at this architectural level is to understand that PlaneGCS does <u>not</u> understand the concept of geometry and constraints used in the Sketcher. For the programmer, it is not aware of what Part::Geometry and Sketcher::Constraint are.

Though this may initially seem as a design flaw or limitation, it is not. Those higher level representations of reality are very important for the remaining operations of the Sketcher, but they are useless when talking about the number of degrees of freedom introduced by a geometric shape, or to calculate the gradients to decide in which direction to move an element when seeking convergence. Similarly, the information used during solving is mostly useless at a higher level. Of course it could have been decided to put everything together, but there would be no advantages and a lot of flexibility would have been lost, for example, the ability of change to another better solver implementation if one ever is created, or to use PlaneGCS in other products (yes there are other products aside from FreeCAD that use PlaneGCS).

Having a higher level Sketcher that understands a representation of the geometry and a solver that uses a different one, makes necessary to bring an interface therebetween in order to connect both. This interface, which we are going to refer to as solver interface, is the class *Sketch*.

Figure 2.1: Sketcher/solver basic architecture



So it shall be clear by now that the Sketcher has three main parts when referring to the solver architecture:

- Sketcher: *SketchObject* class
- Solver interface: *Sketch* class
- Solver: Everything inside the PlaneGCS directory

The latter is the main focus of this document.

2.2 Sketcher Geometry and Constraints

We refer to the higher level geometry and Constraints used at *SketchObject* level.

The Geometry used in the Sketcher is defined in *Geometry.h* and all of it derives from Part::Geometry. The Sketcher has a vector of pointers to such objects, *std::vector<Part::Geometry *>*.

Those Geometry classes are wrappers of OpenCASCADE Geometry classes which add the capability to store the geometry when saving and higher level functions to simplify certain common tasks.

The Constraints used in the Sketcher are defined in *Constraint.h*, not to be confused with *Constraints.h* in the *planegcs* directory. The Sketcher has a vector of pointers to such objects, *std::vector<Sketcher::Constraints *>*.

Those Constraints merely define the relation between parts of the same or different geometric elements. Each constraint has a Type, which is an enum Sketcher::ConstraintType, e.g. Sketcher::Horizontal for an Horizontal constraint.

They accomodate 3 *int* values called First, Second and Third. Each of these is able to store the index within the Geometry array of a Geometry object. There is a Constraint::GeoUndef value that is used when no value is stored there. So, for example, a Horizontal constraint only needs to store one geometry index, the one of the line to which such constraint apply, which will be stored in First. Second and Third will be set to GeoUndef. However, a symmetry constraint will in the most general case require three different geometries indexed.

In addition to those 3 *int* values, a constraint also accomodates 3 *Sketcher::PointPos* enums, FirstPos, SecondPos, ThirdPos, which can take the values *Sketcher::none*, *Sketcher::start*, *Sketcher::end*, *Sketcher::mid*. They default to *Sketcher::none*. With these values, it is possible to indicate that the constraint is to be applied to the edge (*Sketcher::none*), the start point (*Sketcher:start*), the endpoint (*Sketcher::end*) or a center (*Sketcher::mid*). Of course not all geometric elements have all the possible combinations (circles do not have a start or endpoint, a line does not have a center), but all possible combinations define the limits of what this higher layer Constraint can indicate.

Such limitation of three addressable vertices per geometry was discussed at length when introducing the Ellipse and Arc of Ellipse to the Sketcher. The problem with incrementing the number of addressable vertices was twofold, a) a new element could require a variable number of vertices (e.g. B-Spline), b) it would require a substantial change of both the Sketcher and the Sketcher interface. Finally the solution came as a new type of constraint, the Alignment constraint. This type of constraint has a subtype *Sketcher::InternalAlignmentType*, which incorporates a geometry specific subtype, e.g. *Sketcher::EllipseFocus1*, which identifies that a second geometry, in this case a point, is to be aligned to that position of an Ellipse. Notably the geometry is not restricted to a point. It can be a line, as in the case of the Major axis of an Ellipse, or a circle, as in the case of the control points of a B-Spline. This provides a high degree of flexibility in the implementation and in the ability to support complex shapes. However, as it will be apparent in the following sections, the flexibility gained by using the internalalignment constraint comes at the expense of having to write specific alignment code.

2.3 The solver

2.3.1 Parameters

The planeGCS solver is constructed around the concept of *parameters*. Every single geometry has a number of parameters that define it. Each *unconstrained* parameter contributes to one degree of freedom (DoF). For example, a point can be moved in the x and y direction, so it has 2 DoFs. A line has two points that can be moved in the x and y direction each, so it has 4 DoFs.

There is a very important non-obvious aspect to parameters and degrees of freedom that should be noted at this point. Saying that one *unconstrained* parameter contributes to one degree of freedom *is different from* saying that a constrained parameter contributes to none. As it will be explained more in detail below, a constraint is a single equation that mathematically restricts the values of one or more parameters. An *unconstrained* parameter may take *any* value in $\mathbb{R}$. A *constrained* parameter may only take a subconjoint of $\mathbb{R}$. A special case of a *constrained* parameter is when the parameter may only take one value (its value is fixed). It is specially important to note that a *constrained* parameter may still fullfil all the constraints with different values.

Only when all the paramaters of a sketch are constrained by an equal number of constraints, which are neither redundant nor conflicting, may we say that the Sketch is fully constrained.

Let's try to illustrate important concept with an example. Let it be a system having two parameters (p_1 and p_2), for example a sketch with a single point, and only one such equation, three different cases might arise: a) that p_1 is fixed to a value while p_2 is free to move (setting a horizontal distance constraint, where the point moves, when dragged, on the vertical direction), b) that p_1 is fixed to a value while p_2 is free to move (setting a vertical distance constraint, where the point moves, when dragged, in the horizontal direction), c) that the constraint defines a relation of p_1 and p_2, the relation being constrained (a distance constraint to the origin, where the point moves around the origin following the trajectory of a circle when dragged). All these cases have 1 DoF. In the first case only parameter p_1 is constrained and therefore p_2 is *unconstrained.* In the second case only parameter p_2 is constrained and therefore p_1 is *unconstrained.* In the last case, *both* parameters are constrained, albeit not with the sufficient number of constraints so as to lead to 0 DoF.

Internally a parameter is nothing but a pointer to a double, *double *.* As such can take a value in $\mathbb{R}$ with a certain precision.

2.3.2 Solver geometry

As previously anticipated, the planeGCS has a representation of the Geometry which heavily differs from that of the Sketcher. Geometry is defined in *Geo.h.* The aim of these Geometry classes is manyfold, the most important functions are:

[a)]

1. keep track of the *parameters* of each geometry

2. store geometry specific solver information

3. facilitate the creation of the equation systems

4. simplify the definition of more complex geometries

Every geometry directly or indirectly inherits from class *DependentParameters*. This class acts as an interface to feedback solver information relating to each specific geometry. Currently the class has only one member *hasDependentParameters* which defaults to false and is updated by the solver during solving, thereby enabling to identify whether a given geometry has any parameter that is *dependent*. Here *dependent* means parameters that the solver identifies as not having reached that special state of being constrained to a single value.

The simplest geometry class in the planeGCS solver is the *Point*, which unsurprisely stores two separate pointers to double, one for each dimension. Such a point is not only the representation an individual point of the Sketcher but it is heavily used for the defition of more complex geometries at solver level, e.g. line segments or Ellipses.

Before further describing the geometry, it is necessary to introduce a solver-specific vector concept, which has become essential for the definition of solver constraints, the *DeriVector2*. This class stores a bi-dimensional vector together with the partial derivatives with respect to a *given parameter*. Of course, this class may be used as a normal bi-dimensional vector ignoring the derivatives, thereby incurring a memory penalisation. However, it is in the simultaneous and seamless calculation of vector functions and its derivatives where it excels. General vector operations, like normalisation, scalar products, sums, substractions, scaling, rotation and linear combinations are possible. For the moment, it shall be sufficient to understand that a DeriVector2 is a special type of vector capable of indicating a bi-dimensional position or vector together with the derivatives of that vector with respect to a given parameter (i.e. how the vector varies when the value of such parameter changes). The advantages behind this invention will become evident to the reader thoughout this document.

The most important geometry class at solver level is the abstract class *Curve*. Apart from the general principle of reusing code instead of duplicating it, it enforces an important requirement to any other curve deriving from it, i.e. implementing the pure virtual function *CalculateNormal*:

```
virtual DeriVector2 CalculateNormal(Point &p,
double* derivparam = 0) = 0;
```

It may appear at a first glance as a nuance to have to implement such function. However, implementing it results in avoiding to have to implement several constraints, which would use the result of this function in a general routine.

There are another three pure virtual functions:

```
virtual int PushOwnParams(VEC_pD &pvec) = 0;
virtual void ReconstructOnNewPvec (VEC_pD &pvec, int
&cnt) = 0;
virtual Curve* Copy() = 0;
```

These functions facilitate the construction of equation systems and will be explained in sucessive sections.

Additional advantages may be obtained if the programmer provides an implementation for:

```
virtual DeriVector2 Value(double u, double du, double* de-
rivparam = 0);
```

However this is not mandatory. The only rule being that if it is not implemented, it may not be called, as it would result in a failed assert. This is useful for shapes not having a point on object constraint implementation, or a not yet implemented one (at this time B-Splines is a good example of this).

2.3.3 Solver constraints

As in the case of geometry, the planeGCS has a representation of the constraints which heavily differs from that of the Sketcher. In fact, there is not even a one to one relationship between what the Sketcher regards as constraints and what the solver uses to implement them. Constraints are defined in *Constraints.h*.

The aim of the constraint classes it to introduce restrictions while the solver iterates the values of the parameters seeking for a solution. Obviously the solver constraints originate from the sketcher constraints. However while the latter operate on edges and vertices, the former operates on the parameters of the solver geometry.

Generally speaking a Sketcher Constraint is translated into one or more solver constraints. A single solver constraint may take away a maximum of 1 DoF (none if it is redundant).

When referring to determining the properties of the equation system, a considerable part of the solver code is written under the abstraction, well known from Algebra, of using matrices defined in terms of parameters and constraints. The actual form will be presented in sucesive sections with more detail, but it is anticipated that such form is the Jacobian matrix of the system, having as values the gradients of each of the constraints with respect of each of the parameters of the system. This leads to asymilating the number of parameters with the maximum number of degrees of freedom a system may have and constraints with the number of degrees of freedom that have been taken away. Of course, that would only be true if all the constraints were neither redundant nor conflicting. In terms of the matrix such constraints that are neither redundant nor conflicting constitute *linearly independent* relations or equations which give rise to linearly independent rows in matrix from. Given the one to many relationship between Sketcher constraints, that are input by the user and solver constraints, the existence of solver constraints that are redundant is not uncommon. In fact it happens much more often than what the user perceives as a redundant Sketcher constraint, as many of the redundant solver constraints are silently handled by the solver itself. The presence of redundant solver constraints means that the solver has *linearly dependent* relations, equations, rows in the matrix. As a consequence, it is obvious that the number of degrees of freedom will no longer be determined by the difference between the number of paramaters and the number of constraints. However, as known from algebra the number of independent relations of such system may be obtained by calculating the *rank* of the matrix defining the system, and the number of degrees of freedom from the difference between the number of parameters and the rank. At present it is sufficient to understand that constraints act in combination to restrict the values that parameters of the system may take and that the gradients of the constraints with respect to each of the parameters play a most important role in defining such combinations.

The main functionalities of a solver constraint are:
[a)]

1. store sketcher constraint properties, e.g. driving/reference
2. store the parameters it constraints.
3. calculate the error in meeting the constraint.
4. calculate the gradients with respect to a given parameter
5. limit the step during solving.
6. manage constraint priority, e.g. for dragging

Of them, the calculation of *errors* and *gradients* is at the core of the concept of a geometric constraint solver. Writting a constraint is generally not a trivial matter. The basis of a constraint is a mathematical expression identifying an *error* when meeting a constraint.

For example, for a point on an ellipse this expression is used for the error:

$$e = \sqrt{(F_{2,y} - P_y)^2 + (F_{2,x} - P_x)^2} + \sqrt{(F_{1,y} - P_y)^2 + (F_{1,x} - P_x)^2} - 2a \quad (2.1)$$

This error function in based on the principle that sum of the distances from a point (P) of the ellipse to its foci (F_1 and F_2) equals two times the major axis length (a). Therefore, given a point P that is not on the ellipse, there will be an error, which will be larger the further away the point is from the ellipse curve. As such the error is a magnitude that indicates how well or bad the geometry is converging to the right solution for that constraint.

The main limitation is that an error function does not allow by itself to determine in which direction the parameters should move in order to reduce that error. This is the reason of introducing the gradient of the constraint. For a given constraint there are as many partial derivatives constituting the gradient as parameters intervene in its definition.

For the case of point on ellipse above, the intervening parameters are: the two coordinates of the point (P_x and P_y), the coordinates of the left focus of the ellipse ($F_{1,x}$ and $F_{1,y}$), the coordinates of the center point of the ellipse (C_x and C_y), the length of the minor axis of the ellipse (r). It is noted that other parameters could have been used to define an ellipse, but those above were the ones that finally were implemented. This means that seven different partial derivatives must be calculated only for the point on object constraint. This is just one of them, the one corresponding to the length of the minor axis of the ellipse, which was selected for its conciseness with respect to the others:

$$\frac{\partial e}{\partial r} = \frac{-2r}{\sqrt{r^2 + (F_{1,x} - C_x)^2 + (F_{1,y} - C_y)^2}} \quad (2.2)$$

After this incursion into the constraint inner workings, it may not come to a surprise that the most important member functions of a constraint are:

```
virtual double error();
virtual double grad(double *);
virtual double maxStep(MAP_pD_D &dir, double lim=1.);
```

The last function may be used to limit the step during convergence if stability requires a lower speed of convergence, but it is seldom implemented.

Because manually calculating differentials is cumbersome, error prone, time consuming and it leads to non-obvious formulas that cannot be inmediatly understood by a human, DeriVector2 was created.

After DeriVector2 was introduced not a single constraint has been written using manual formulas as shown above and some of those which were already written and had issues have been rewritten using DeriVector2. Using DeriVector2 error and gradient are calculated using a same function, usually called *errorgrad*, which is then called from the *error* and *grad* functions passing the right parameters.

A very good example is *ConstraintEllipseTangentLine::errorgrad*. The error function is based on the property that the distance between one focus of the ellipse and the position of the other focus mirrored about a line tangent to an ellipse has a given length of 2a.

```
DeriVector2 p1 (l.p1, param);
DeriVector2 p2 (l.p2, param);
DeriVector2 f1 (e.focus1, param);
DeriVector2 c (e.center, param);
DeriVector2 f2 = c.linCombi(2.0, f1, -1.0); // 2*cv - f1v

//mirror F1 against the line
DeriVector2 nl = l.CalculateNormal(l.p1, param)
                .getNormalized();

double distF1L = 0, ddistF1L = 0; //distance F1 to line

distF1L = f1.subtr(p1).scalarProd(nl,&ddistF1L);

//f1m = f1 mirrored
DeriVector2 f1m = f1.sum(nl.multD(-2*distF1L,
                                  -2*ddistF1L));

//calculate distance form f1m to f2
double distF1mF2, ddistF1mF2;
distF1mF2 = f2.subtr(f1m).length(ddistF1mF2);

//calculate major radius (to compare the distance to)
double dradmin = (param == e.radmin) ? 1.0 : 0.0;
double radmaj, dradmaj;
radmaj = e.getRadMaj(c,f1,*e.radmin, dradmin, dradmaj);
```

```
if (err)
    *err = distF1mF2 - 2*radmaj;
if (grad)
    *grad = ddistF1mF2 - 2*dradmaj;
```

Gradients depend on the parameter that is being evaluated by the solver, so the first step is to calculate the positions of the endpoints of the line to be tangent to the ellipse and how this positions vary with changes in the parameter under evaluation, i.e. the partial derivatives with respect to the parameter. Similary, it is calculated how the left focus, the center of the ellipse, or the right focus varies with the parameter. While the former ones are parameters of the line and ellipse, the latter, the right focus, is already defined when one knows the remaining parameters and as such is not one of the parameters used to define the ellipse. For this reason, it is calculated from the center of the ellipse and the left focus using a linear combination of those points.

Next a normal vector to the line at the start point is obtained, including the partial derivatives. Then vectorial calculus is applied to this information to obtain the error of an expression with the current value of the parameter, and the gradient. The gradient enables the solver to know how the parameter shall be modified in order to minimise the error.

It is remarkable that both error and gradient can be calculated simultaneously using *DeriVector2* and how the minimization is implemented can still be followed by a human.

2.3.4 System and Subsystems

The core of the solver is the class *GCS::System* in *GCS.h*. It contains all the *parameters* and *constraints*, and notably the decoupling of *parameters* into components. These components define subconjoints of parameters and constraints that may be solved independently. This components are the subsystems.

There are two types of subsystems within GCS: a) *subsystems* originating from constraints without priority, and b) *subsystemsAux* originating from constraints with priority.

The priority is used in certain operations involving augmented solutions or programatic movements of geometry.

The *GCS::Subsystem* class in *SubSystem.h* performs several calculation operations on the *parameters* and *constraints* of the subsystem, like for example, the calculation of residuals, jacobi or gradients. These calculations are used by the solver algorithms defined in *GCS:System*, e.g. DogLeg.

2.4 Solver interface

Taking into account the big differences between Sketcher, *SketchObject*, geometry and contraints and those of the solver, it is not surprising that there is an intermediary that handles the bidirectional interation between the two of them.

While it would be possible to include here a higher degree of detail about the interface, out of pedagogical reasons it will be deferred until the Sketcher functionality that the solver supports is presented.

Chapter 3

Sketcher functions

3.1 Introduction

While it is well possible to understand the lower solver algorithms without considering the functionality that the solver offers to the Sketcher, introducing the functionality beforehand helps to better understand the architecture chosen for the solver interface and the reasons behind some of the solver features.

All the functionality described in this chapter is implemented in *SketchObject* in *SketchObject.h*.

3.2 Modifications of the Sketch

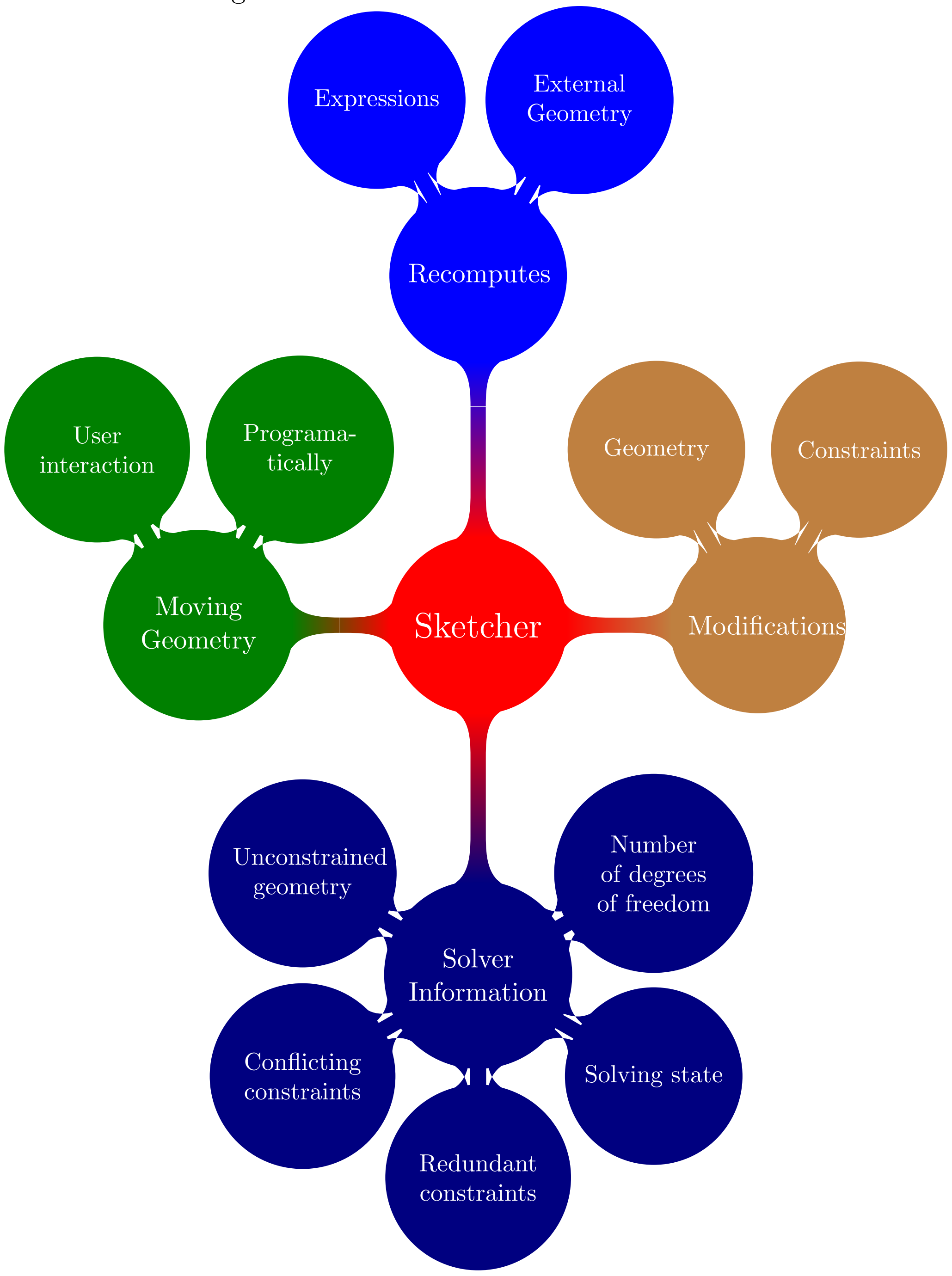
It is the most important functionality of the Sketcher, its ability to *adapt* the geometry to fulfill some Sketcher constraints set by the user. One could think of how one operates while being in edit mode in the sketcher.

In fact, there are two possibilities for solving modifications introduced while the sketcher is in edit mode: a) solving and b) recomputing. Thinking on the user interface, this is regulated by the *Auto-Update* checkbox in the *solver messages* task panel. When it is checked, a recompute is triggered, otherwise a solve.

The main difference is that the recompute will propagate all the changes to the sketch to any object outside the sketcher that depends on it, while the solve will not. As a result, a solve will not change the geometry of the sketcher as seen from outside edit mode. Of course, a recompute comprises a solve, but what is relevant here is that they are separate functions.

The c++ function implementing the *solve* functionality is:

Figure 3.1: Sketcher solver related functions



```
int solve(bool updateGeoAfterSolving=true);
```

A call to this function involves:
[(i)]

1. Resets any ongoing dragging or programatical geometry movement procedure.

   ```
   solvedSketch.resetInitMove();
   ```

2. Transfer the current values of Geometry, Constraints and ExternalGeometry to the solver interface and requests a *diagnosis* of the equation system. This evaluation returns among others, the Degree of Freedom of the system and whether conflicting or redundant constraints are present. If there are redundant or conflicting constraints, or otherwise the system is overconstrained, the solving process is stopped.

   ```
   lastDoF = solvedSketch.setUpSketch(
                  getCompleteGeometry(),
                  Constraints.getValues(),
                  getExternalGeometryCount());

   lastHasConflict=solvedSketch.hasConflicts();
   lastHasRedundancies=solvedSketch.hasRedundancies();
   lastConflicting=solvedSketch.getConflicting();
   lastRedundant=solvedSketch.getRedundant();
   ```

   (a) The equations system is solved and the status retrieved. However, if the solver failed, the process continues.

   ```
   lastSolverStatus=solvedSketch.solve();
   ```

   (b) If the solver did not fail and the parameter *updateGeoAfterSolving* is set, the geometry of the sketch is updated based on the result of the solver.

The recompute is treated in the following section together with the other cases that trigger a recompute.

3.3 Recomputes

When a property of an object changes, the object might have been changed and appropriate action has to be taken. The mechanism used in FreeCAD is called *recompute* and the function that is executed upon recompute is:

```
App::DocumentObjectExecReturn *execute(void);
```

A call to this function involves:

[(i)]

(a) Any external geometry might have changed, so the edit mode representation is updated. If such update would fail any constraint to external geometry will be removed.

```
rebuildExternalGeometry();
```

(b) A solve with *updateGeoAfterSolving* set is executed. Any error originated, let it be redundant or conflicting constraints or otherwise overconstrained, or actual solver failure is notified to the user.

```
int err = this->solve(true);
```

(c) In case of no errors, the bi-dimensional shape of the sketch is updated with the result obtained by the solver.

```
Shape.setValue(solvedSketch.toShape());
```

The sketcher itself may trigger recomputes during edit mode in *Auto-Update* mode, as described in section 3.2. This directly propagates all the changes in the sketch to other objects depending on it.

Similarly, if a sketch depends on expressions or on external geometry and they change, a recompute of the sketch is generated.

3.4 Moving sketch geometry

Sometimes it is necessary to move sketch geometry even though the solver does not require such moves in order to fulfil the constraints. This moving functionality may be the result of *user interation*. One example is the operation of dragging geometry. Moving functionality is also used when the need arises to *programatically* move a given geometry or one of its defined vertices. One example relates to the creation of a fillet in the sketcher. When creating a fillet it is necessary to programatically move the endpoints of the preexisting geometry to which the fillet is to be applied, to the points of start and end of the arc created to define the fillet. This must be done before any solving is performed, otherwise the solver will move both the arc and the geometry freely, resulting in unwanted behaviour.

While the two previous functionalities directly or indirectly rely on the solving functionality of the sketcher seen before:

```
int solve(bool updateGeoAfterSolving=true);
```

programatic moves of vertices is achieved using the *SketchObject* function:

```
int movePoint(int GeoId, PointPos PosId,
              const Base::Vector3d& toPoint,
              bool relative=false,
              bool updateGeoBeforeMoving=false);
```

A call of this function:

[(i)]

(a) If an update of the geometry is necessary before moving according to parameter *updateGeoBeforeMoving*, solver geometry is updated. Otherwise, the geometry available at solver level is used for the moving operation.

(b) If the last execution of the solver returned an overconstrained sketch or one having conflicting constraints the moving of geometry is aborted.

(c) The solver interface point moving functionality is called. This functionality internal performs a special kind of solving, which returns a solving status.

```
lastSolverStatus = solvedSketch.movePoint(GeoId,
                   PosId, toPoint, relative);
```

(d) If this special solve succeeded, the geometry of the sketcher is updated based on the solution found.

(e) The solver interface moving functionality is reset as the moving operation has finished.

```
solvedSketch.resetInitMove();
```

In the case of user interation, it uses a similar interface as even if it responds to user interaction, as in the end it is nothing else than a programatic move according to the input of the user. However, there are two main meaningful differences: a) it is interfaced directly from the ViewProvider, b) it uses an additional method of the *solver interface*.

The first difference is an obvious consequence of user interaction. The second one is a consequence of an ongoing move operation that requires intermediate updates. In the particular case of the drag and drop, in order to redraw during the dragging process and give the impression of interaction.

An example of such dynamic operation can be found in *ViewProvider-Sketch.h*:

```
bool ViewProviderSketch::mouseMove(
                              const SbVec2s &cursorPos,
                              Gui::View3DInventorViewer
*viewer)
```

It is not worth to go into this function in detail, but it is important to identify that such operation starts with a call to the following function of the solver interface:

```
getSketchObject()->getSolvedSketch().initMove(GeoId,
                              PosId, false);
```

Subsequent calls are directly to the solver interface:

```
getSketchObject()->getSolvedSketch().movePoint(GeoId,
                              PosId, vec, false)
```

However, when the drag and drop operation ends, the final movement is effected via *SketchObject* in order to update the Sketcher geometry with the solver geometry.

3.5 Sketcher relevant solver information

There are two types of information returned by the solver: a) general information and b) specific information. The first type is obtained every time the sketcher interface is triggered to set up a sketch or solve it. The second type is obtained only on demand after a full solving operation has finished.

As it has been presented already in section 3.2, the solver must be first given the geometry it has to solve, which is performed using the *setUpSketch* function of the solver interface. This function performs an initial evaluation or diagnosis of the equation system. Important information is obtained from this initial evaluation: a) the number of degrees of freedom of the system, b) whether there are redundant constraint and which ones they are, c) whether there are conflicting constraints and which ones they are.

This information enables the Sketcher to know if it is worth trying to actually solve the system or not. However it also informs the user about the number of degrees present and which problems regarding constraints must be tackled in order to achieve a valid geometric solution.

```
lastDoF = solvedSketch.setUpSketch(
                getCompleteGeometry(),
                Constraints.getValues(),
                getExternalGeometryCount());

lastHasConflict=solvedSketch.hasConflicts();
lastHasRedundancies=solvedSketch.hasRedundancies();
lastConflicting=solvedSketch.getConflicting();
lastRedundant=solvedSketch.getRedundant();
```

If the diagnosis shows no problem, then the actual solving is performed and additional information is obtained:

```
lastSolverStatus = solvedSketch.solve();
```

This status information is defined in *GCS.h* and may take several values:

- **GCS::Success**: Solution zeroes the error function

- **GCS::Converged**: Solution minimizes the error function, but does not zero it.
- **GCS::Failed**: The solver failed to find a solution.
- **GCS::SuccessfulSolutionInvalid**: The solver found a solution, but this solution is not accepted by OpenCASCADE.

In practice this information is currently used only in binary form at Sketcher level, either *GCS::Success* or *GCS::Failed*. This information is fed back to the user in the *solver messages* task panel.

Regarding the specific information, currently the solver is capable of identifying geometry having parameters that are dependent within the definition of sections 2.3.2 and 2.3.3. However, this cannot be achieved with the faster algorithm that the solver uses by default for QR decomposition of sparse matrices, *SparseQR*. Therefore in order to achieve it a specific solve is requested with a full pivoting dense QR decomposition. Asuming that *Obj* below is an instance of *SketchObject*, this specific solve may be requested by manually setting the QR algorithm:

```
Obj->getSolvedSketch()
      .setQRAlgorithm(GCS::EigenDenseQR);
Obj->solve(false);
Obj->getSolvedSketch()
      .setQRAlgorithm(GCS::EigenSparseQR);
```

After that solve, the solver interface may be queried for a given geometry index, *geoId* and a given Sketcher::PointPos, *pos*, as follows:

```
Obj->getSolvedSketch()
      .hasDependentParameters(geoId, pos);
```

A complete example of how to obtain this information is available in *CommandSketcherTools.cpp*:

```
void CmdSketcherSelectElementsWithD-
oFs::activated(int)
```

Alternatively, *SketchObject* also has a function that returns a vector of geo index and PointPos pairs containing all the dependent parameters. This is also accessible from Python.

void
SketchObject::getGeometryWithDependentParameters
(std::vector<std::pair<int,PointPos>>& geome-
trymap)

Chapter 4

Solver interface

4.1 Introduction

The most important function of the solver interface is that of interfacing between the sketcher (*SketchObject*) and the solver (*PlaneGCS*). This mainly involves translating the geometry and constraint information from its Sketcher form to a form usable by the solver before solving it and conveying the result embedded in the solver geometry back to the Sketcher. In order to effectively achieve this function, the solver interface must keep track of sketcher and solver, for which it maintains a plurality of *data structures*.

As it has already been introduced in chapter 3, the solver provides interfaces to enable: a) solving, b) moving geometry and c) retrieving solver information.

4.2 Data structures

4.2.1 Geometry

The geometry at Solver interface level is structured into a vector of a solver interface defined type, *GeoDef* and a plurality of geometry type specific vectors containing the solver representation of each geometry:

```
std::vector<GeoDef> Geoms;

std::vector<GCS::Point> Points;
```

```
std::vector<GCS::Line> Lines;
std::vector<GCS::Arc> Arcs;
std::vector<GCS::Circle> Circles;
std::vector<GCS::Ellipse> Ellipses;
std::vector<GCS::ArcOfEllipse> ArcsOfEllipse;
std::vector<GCS::ArcOfHyperbola>      ArcsOfHyper-
bola;
std::vector<GCS::ArcOfParabola> ArcsOfParabola;
std::vector<GCS::BSpline> BSplines;
```

The solver interface defined type is a struct including a pointer to a copy of the Sketcher geometry, a solver interface defined geometry type, a flag to indicate whether the geometry is external or not, an index that corresponds to the position of this geometry within the geometry type specific vectors containing the solver representation, and an index for each of the three possible addressable vertices of a shape in the vector containing the solver representation of points.

```
Part::Geometry * geo;
GeoType type;
bool external;
int index;
int startPointId;
int midPointId;
int endPointId;
```

With this information the Sketcher keeps track of the geometry and enables the bidirectional synchronisation of geometry information.

4.2.2 Constraints

The constraints at Solver interface level are structured into a vector of a solver interface defined type, *ConstrDef*.

```
std::vector<ConstrDef> Constrs;
```

The solver interface defined type including a pointer the sketcher constraint, a flag indicating whether the constraint is driving or reference, a first pointer to double pointing to the memory address of a datum value of the constraint provided it has one, and a second pointer to double pointing to the memory address of a second datum value provided it has one. The latter is only used in the case of a Snell's law constraint.

```
Constraint * constr;
bool driving;
double * value;
double * secondvalue;
```

4.2.3 Parameters

As it was introduced in section 2.3.1, the solver works based on parameters. As the solver geometry is created at the solver interface, another important function of the sketcher interface is allocating memory for these parameters and keeping track of them.

There is a first vector of pointers to double that contains all the parameters of the system that are not fixed (e.g. the parameters of external geometry), a second vector contains only those which are driven (i.e. the datums of reference constraints), a third one contains only fixed parameters (e.g. the parameters of external geometry), a fourth one contains those parameters that, after a diagnosis using the full pivoting dense QR decomposition, have been detected as being dependent parameters. Then there are two additional arrays of doubles, not pointers to doubles, which store parameters used for programatically moving geometry.

```
std::vector<double *> Parameters;
std::vector<double *> DrivenParameters;
std::vector<double *> FixParameters;
std::vector<double *> pconstraintplistOut;
std::vector<double> MoveParameters, InitParameters;
```

4.3 Transfer from the sketcher to the solver

As presented in previous sections, the geometry of the Sketcher is transferred to the solver using the solver interface function:

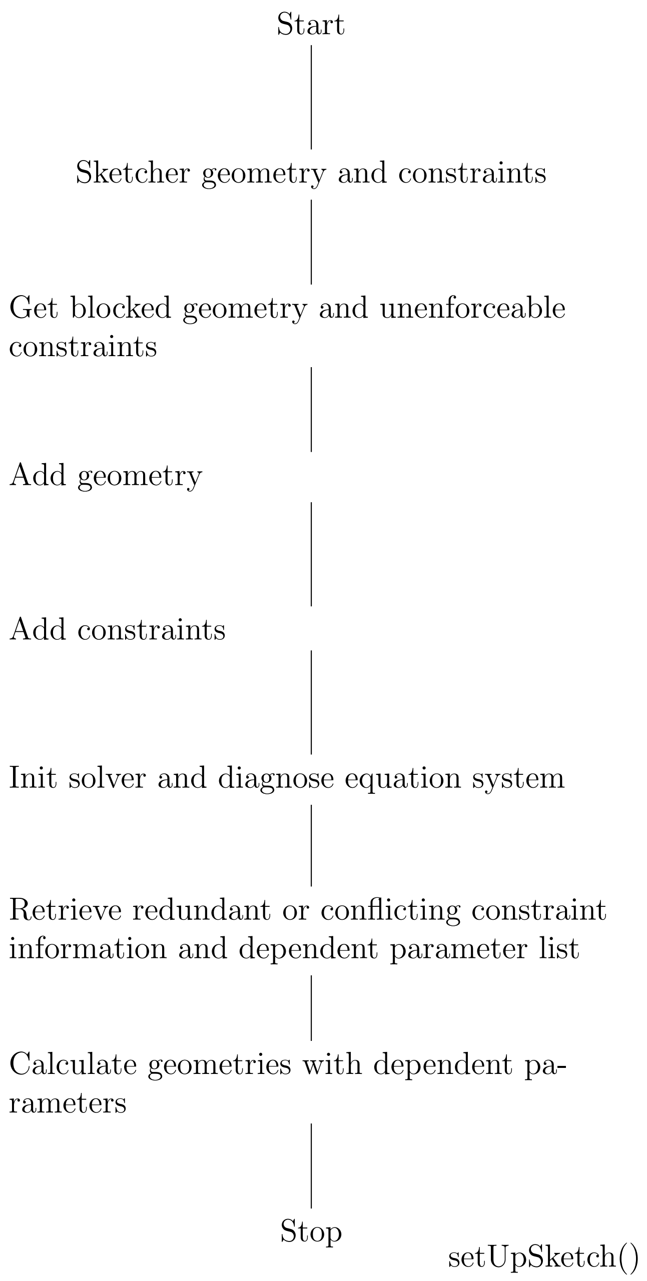


Figure 4.1: Flow chart of setting up sketch
Start
Sketcher geometry and constraints
Get blocked geometry and unenforceable constraints
Add geometry
Add constraints
Init solver and diagnose equation system
Retrieve redundant or conflicting constraint information and dependent parameter list
Calculate geometries with dependent parameters
Stop
setUpSketch()

```
int setUpSketch(
    const std::vector<Part::Geometry *> &GeoList,
    const std::vector<Constraint *> &ConstraintList,
    int extGeoCount=0);
```

The sketcher geometry is *complete* geometry. This means that it includes both internal geometry and external geometry in the same vector. The internal geometry is pushed into the vector first and then the external geometry. The last parameter indicates how many external geometries are present in the vector. This allows to calculate the boundary between internal and external geometry.

As this is the point at which geometry and constraints are added to the solver, it is also the point in which it may be decided which geometries and constraints to send to the solver. In other words, if for some reason a geometry or a constraint shall not take part in the solving procedure or shall take part but in a special way, it needs to be processed accordingly at this point. One such example is the implementation of the *Block* constraint. Which sets the parameters of the blocked geometry as if they were external geometry, i.e. fixed, and excludes some other constraints as a result.

The function to add geometry iterates the vector of geometry taking into account any blocked geometry, checks for the Sketcher type and executes a type specific addition function. A simple example illustrating this process corresponds to function:

```
int Sketch::addLineSegment(
    const Part::GeomLineSegment &lineSegment, bool
fixed)
```

The following is an example of such a specific function for a line segment. First depending on whether the geometry is external or internal, the right vector of parameter is selected. Second, a copy of the geometry is created. A solver interface defined type, *GeoDef*, object is created and the pointer and type data assigned. Corresponding solver geometry types are created of the Points and the Line, and memory is allocated in the relevant parameters vector for the coordinates of the start and endpoints. Points are added to the points vector and the line to the lines vector. The indices of the start and endpoints in the Points array are added to the *GeoDef* object, which is added to the sketcher interface vector containing all the geometries.

```
std::vector<double *>&params =
    fixed ? FixParameters : Parameters;

GeomLineSegment *lineSeg =
    static_cast<GeomLineSegment*>(lineSegment.clone());

GeoDef def;
def.geo = lineSeg;
def.type = Line;

Base::Vector3d start = lineSeg->getStartPoint();
Base::Vector3d end = lineSeg->getEndPoint();

GCS::Point p1, p2;

params.push_back(new double(start.x));
params.push_back(new double(start.y));
p1.x = params[params.size()-2];
p1.y = params[params.size()-1];

params.push_back(new double(end.x));
params.push_back(new double(end.y));
p2.x = params[params.size()-2];
p2.y = params[params.size()-1];

def.startPointId = Points.size();
def.endPointId = Points.size()+1;
Points.push_back(p1);
Points.push_back(p2);

GCS::Line l;
l.p1 = p1;
l.p2 = p2;
def.index = Lines.size();
Lines.push_back(l);

Geoms.push_back(def);

return Geoms.size()-1;
```

The function to add constraints iterates the vector of constraints taking into account any unenforceable constraint. For each sketcher constraint it creates a solver interface defined type, *ConstrDef*, object, assignes the constraint pointer and sets the driving flag accordingly. Then, the constraint type is checked. If the constraint contains datums, memory is allocated for a parameter and the value of the constraint is assigned, the parameter is assigned to the *FixParameters* vector if the constraint is driving and to the *Parameters* and *DrivenParameters* vectors if it is reference. This allows to enforce the constraint value in case it is driving and to let it be modified by the solver if it is reference. Datum or not, a constraint specific function is called to create the one or more corresponding solver constraints. An example of such a function is:

```
int Sketch::addDistanceXConstraint(int geoId,
                                    double * value, bool
driving)
```

In this specific case, this constraint the horizontal distance constraint is implemented via a solver difference constraint. First it is checked that the geo index of the constraint is valid and corresponds to the right geometry type. The line is retrieved from the Lines vector using the index stored in the *GeoDef* vector. A tag is created. The tag stored in the solver constraint. In a case that several solver constraints are necessary to implement the Sketcher constraint, all the solver constraints carry the same tag. This allows to map the findings of the solver for its constraints to the sketcher constraints.

```
geoId = checkGeoId(geoId);

if (Geoms[geoId].type != Line)
    return -1;
GCS::Line &l = Lines[Geoms[geoId].index];

int tag = ++ConstraintsCounter;
GCSsys.addConstraintDifference(l.p1.x, l.p2.x, value,
tag, driving);
return ConstraintsCounter;
```

Once geometry and constraints are added to the solver, the solver interface performs an initilisation and diagnosis:

```
GCSsys.clearByTag(-1);
GCSsys.declareUnknowns(Parameters);
GCSsys.declareDrivenParams(DrivenParameters);
GCSsys.initSolution(defaultSolverRedundant);
```

The actual functionality provided by these solver commands will be explained in the chapter dedicated to the PlaneGCS.

4.4 Solve

The solve function of the solver interface is executed directly from the sketcher. However it is also executed by the solver interface functions implementing the programatic movement of geometry. The function is defined in *sketch.cpp*:

```
int Sketch::solve(void)
```

The solver includes a plurality of algorithms: a) DogLeg, b) BFGS, c) Levenberg-Marquardt and d) SQP.

From that plurality of algorithms the preferred one is executed. If it succeedes, the geometry of the sketch is updated. If not the remaining algorithms are fired one after the other until one succeedes or none does.

4.5 Transfer from the solver to the sketcher

Upon successful solve the geometry of the solver, the solution is applied and the geometry of the Sketcher is updated with the solution. If the update is sucessful the values of the non driving constraints need to be updated to the values found by the solver. The solution may be invalid because OpenCASCADE may not accept it. In such a case the solve and geometry are reverted.

```
if (ret == GCS::Success) {
    GCSsys.applySolution();
     valid_solution = updateGeometry();
     if (!valid_solution) {
         GCSsys.undoSolution();
```

```
        updateGeometry();
    }
    else {
        updateNonDrivingConstraints();
    }
}
```

The update of the geometry is done by iterating the *GeoDef* vector of the solver interface and using the indices to retrieve the parameters and assign them to the copy of the Sketcher geometry in the *GeoDef* structure. The geometry assignment is performed at *SketchObject* level, using the function *extractGeometry* to obtain the copy of the Sketcher geometry. The reason for this copy and assignment is twofold: a) so that the integrity of the sketcher geometry is independent of the potential solver inability to reach a solution, and b) because the Property in *SketchObject* must be set so that the change triggers and the property history used in the undo function are properly maintained.

4.6 Programatic Movement of geometry

As it has been explained in section 3.4, the functions of the *solver interface* dealing with programatic movement of geometry are:

```
int Sketch::initMove(int geoId, PointPos pos, bool fine)
void Sketch::resetInitMove()
int Sketch::movePoint(int geoId, PointPos pos,
                        Base::Vector3d toPoint, bool
relative)
```

The *initMove* function identifies the part of the geometry to move via the geo index *geoId* and the PointPos enum. The function creates new temporary*solver* constraints which are given priority when finding a solution. These constraints serve the purpose of forcing a vertex or point within an edge to follow the position or vector that will be given via the *movePoint*.

A simple example to illustrate what the *initMove* function does follows.

```
if (Geoms[geoId].type == Point) {
   if (pos == start) {
        GCS::Point                &point                =
Points[Geoms[geoId].startPointId];
        GCS::Point p0;
        MoveParameters.resize(2);
        p0.x = &MoveParameters[0];
        p0.y = &MoveParameters[1];
        *p0.x = *point.x;
        *p0.y = *point.y;
        GCSsys.addConstraintP2PCoincident(p0,point,-
1);
    }
}
```

A point geometry may only have a part, the PointPos Sketcher::start. The point with its *parameters* is retrieved from the *solver* geometry. A new point, *p*0 is created and the parameters mad to point to the *MoveParameters* vector. The values of these parameters are initialised to the same value as the point to move. Then a new *solver* constraint is created to indicate the *solver* that these two points should be at the same position. It is noted that the constraint is added with a negative tag value of -1. This is to indicate that this constraint takes priority over any other constraint. As it will be explained, the solver uses a special *algorithm* to solve systems having priority constraints.

The *initMove* function ends with the statements:

```
InitParameters = MoveParameters;
GCSsys.initSolution();
isInitMove = true;
```

The relevance of them will be more apparent when presenting the *solver initSolution* function. At this point it suffices to note that it initilises the equation system so as to enable the *solver* algorithms to act on them.

The actual movement of geometry is effected by the *movePoint* function. Two cases are separated here, a single programatic movement of geometry and an interactive one. For a single programatic movement, there is no need to call the *initMove* function above, because *movePoint* checks whether the point is initialised. If it is not, it calls the *initMove* function to initilise it.

However, in the case of interactive movement, it is not desirable to initilise the solution on every movement call as it leads to unwanted results, e.g. lagging. In this case, a single solution initialisation is updated every time the movement distance from the initial point overpasses 20 times the distance initial movement, i.e. the distance indicated by the first call to the *movePoint*. This mechanism prevents jumping during dragging operations.

There are two types of movement: a) the *relative* movement, in which the parameter *toPoint* indicates the vector of movement from the original position, which is mostly using for *interactive* movements, like dragging, b) the *absolute* movement, in which the parameter *toPoint* provides the position to which the initial point shall be moved.

The internal workings are illustrated in the following example:

```
if (relative) {
 for (int i=0; i < int(MoveParameters.size()-1); i+=2)
{
   MoveParameters[i] = InitParameters[i] + toPoint.x;
   MoveParameters[i+1] = InitParameters[i+1] + to-
Point.y;
 }
} else if (Geoms[geoId].type == Point) {
 if (pos == start) {
   MoveParameters[0] = toPoint.x;
   MoveParameters[1] = toPoint.y;
 }
}
```

The *MoveParameters* vector is updated with the new position to which the point should move, in either a *relative* or *absolute* way.

Expectedly, the function ends with a *solve* in order to enforce the constraint created to the new position.

It is not possible for the *solver interface* to know when an interactive moving operation is finished. For this reason, the function *resetInitMove* is called at *Sketcher* level when operation is over.

4.7 Retrieving solver information

Most of the *solver information* is obtained directly from the solver itself and the solver interface acts merely as buffer. This is the case of *redundant* constraints, *conflicting* constraints, *degrees of freedom*. The buffer is relevant in that one can operate several *solver* level solvings and only updating this information when it is relevant for state of the sketch. For example, if constraints are added artificially for certain operations that solver information would be worthless for the user.

Most of the diagnosis information is obtained from the solver diagnosis triggered in function *setUpSketch*. This is the case of *redundant* constraints, *conflicting* constraints, *degrees of freedom* and *dependent parameters*.

```
int setUpSketch(
    const std::vector<Part::Geometry *> &GeoList,
    const std::vector<Constraint *> &ConstraintList,
    int extGeoCount=0);
```

Regarding the geometry having parameters that are dependent within the definition of sections 2.3.2 and 2.3.3, the solver interface does perform a processing, in order to convert the pure *parameter* information available at the *solver* into *sketcher* geometry information.

```
bool hasDependentParameters(int geoId, PointPos
pos) const void calculateDependentParametersEle-
ments(void)
```

The function *calculateDependentParametersElements* iterates the geometry structures at *solver interface* level, *std::vector<GeoDef> Geoms*. For each geometry it looks if the *parameters* identified by the solver as dependent belong to the geometry, and sets the flag *hasDependentParameters* accordingly. This maps the individual parameters to the geometry elements at *solver interface* level that contain them. In addition to that, it further identifies the vertices of each geometry addressable via *PointPos*, and sets the vertices as dependent according to the *parameters* of the vertices. This enables to have information on geometric elements and the vertices separatedly.

The function *hasDependentParameters* relies on this calculation to map these *solver interface* level geometry to actual *Sketcher* geometry. The function returns *true* or *false* whether the *Sketcher* geometry identified by the *GeoId* index and the *PointPos*.

Chapter 5

PlaneGCS

5.1 Introduction

The sketcher interfaces with the solver interface, which in turn interfaces with the solver. The solver holds all solver constraints and information about partitioning them into subsystems and other solution strategies, and is responsible for applying those strategies so as to obtain a valid solution.

5.2 Data structures

5.2.1 Introduction

From a data member point of view, *SketchObject*, defined in *SketchObject.h*, has a data member which is of solver interface type, i.e. *sketch*, as defined in *sketch.h*. In turn the solver interface type, *sketch*, has a data member of solver type *GCS::System* as defined in *GCS.h*.

The most important data structures of the solver are presented below.

When dealing with constraints, the solver constraints are stored in a vector of pointers to those constraints. Upon pushing a new constraint to the constraint vector, two adjacency lists are created, so that given a pointer to the constraint the parameters can be obtained, or given a parameter, the constraints involved.

```
std::vector<Constraint *> clist;
std::map<Constraint *,VEC_pD > c2p;
std::map<double *,std::vector<Constraint *> > p2c;
```

The vector *plist* points to the list of parameters of the solver. The vector *pdrivenlist* points to the subset of parameters of the solver that are the datums of reference constraints. They are set from the solver interface using the function *declareDrivenParams*.

When dealing with parameters, certain operations take advantage of a map of parameters *pIndex*, in which parameters act as key and the map returns the index in the parameters vector of the given parameter. This map is generated by the *declareUnknowns* function.

Before undertaking any solving operation, the solver stores the values of the parameters in a vector of doubles *reference* using the function *setReferences*. This allows to revert the parameters to the reference values using the function *resetToReference*.

```
VEC_pD plist;
VEC_pD pdrivenlist
MAP_pD_I pIndex;
VEC_D reference;
```

5.2.2 Constraint structure

Constraints are the basis of the solver. In section 2.3.3 the core solving functionality of solver constraints was introduced. However, several aspects regarding the general structure of the constraints will be presented here.

All solver constraints derive from the Constraint class, which is defined as follows:

```
class Constraint
{
protected:
    VEC_pD origpvec;
    VEC_pD pvec;
    double scale;
    int tag;
    bool pvecChangedFlag;
    bool driving;
public:
    Constraint();
    virtual  Constraint(){}

    inline VEC_pD params() { return pvec; }

    void redirectParams(MAP_pD_pD redirectionmap);
    void revertParams();
    void setTag(int tagId) { tag = tagId; }
    int getTag() { return tag; }

    void setDriving(bool isdriving) { driving = isdriving;
}
    bool isDriving() const { return driving; }

    virtual ConstraintType getTypeId();
    virtual void rescale(double coef=1.);
    virtual double error();
    virtual double grad(double *);
    virtual double maxStep(MAP_pD_D &dir, double
lim=1.);
    int findParamInPvec(double* param);
};
```

A constraint stores a first set of parameters of the constraint with the original values of the parameters, *origpvec* and a second set of parameters with the current values, *pvec.* The first set is only used as reference for reverting or redirecting the current values in the second set.

A constraint also has a scale, used to scale the error and gradients. This scale may be modified using the *rescale* method.

The *tag* is an index used to map solver constraints to sketcher level constraints. Additionally, negative values are used for priority constraints which behave specially and are not considered during the diagnosis of the equation system. Such priority constraints are used for example for programatically moving the geometry. It may be interfaced via the *setTag* and *getTag* functions.

The flag *pvecChangedFlag* indicates that the *pvec* vector has changed and any saved geometry pointers in the class shall be reconstructed.

The flag *driving* indicates whether the constraint is driven or reference. It may be interfaced via the *setDriving* and *isDriving* functions.

The function *params* returns a vector to double pointers containing all the constraint parameters.

The constraint has a function to return its type, which is one of the values of the enum *GCS::ConstraintType*.

The function *findParamInPvec* unsurprisingly searches for a given parameter in the parameters of the constraint, and returns the index of the first occurrence, and -1 if not found.

The function *revertParams* resets the parameters vector to the original parameters.

The function *redirectParams* changes the parameters of the constraint with other parameters according to a map. Basically, it searches the parameters of the constraint as key in the map and if there is a coincidence the parameter is assigned associated the parameter in the map.

A practical example of a constraint implementation reuses the basic functionality of the parent class and only defines a few parameters.

```
class ConstraintDifference : public Constraint
{
private:
    inline double* param1() { return pvec[0]; }
    inline double* param2() { return pvec[1]; }
    inline double* difference() { return pvec[2]; }
public:
    ConstraintDifference(double *p1, double *p2, double
*d);
    virtual ConstraintType getTypeId();
    virtual void rescale(double coef=1.);
    virtual double error();
    virtual double grad(double *);
};
```

The constraint uses three parameters, *p1*, *p2* and *d*. Its goal is to maintain a difference between those two parameters.

The constructor basically assigns the parameters to the *pvec* and initializes *origpvec* to the same parameters. The member functions above are just named accessors to the parameters.

Given the simplicity of the constraint, it is interesting to look into the error and gradient functions:

```
double ConstraintDifference::error()
{
   return scale * (*param2() - *param1() - *differ-
ence());
}

double ConstraintDifference::grad(double *param)
{
   double deriv=0.;
   if (param == param1()) deriv += -1;
   if (param == param2()) deriv += 1;
   if (param == difference()) deriv += -1;
   return scale * deriv;
}
```

5.3 Transfer from the solver interface to the solver

5.3.1 Introduction

It was already introduced in section 4.3 how the geometry and constraints were transferred from the sketcher to the solver interface. While the geometry was allocated into *parameters* fully at solver interface level, the transfer of the constraints relied on specific functions of the solver itself.

5.3.2 Constraint creation

In the example presented above, the function responsible for adding a given constraint to the solver was implemented as:

```
geoId = checkGeoId(geoId);

if (Geoms[geoId].type != Line)
     return -1;
```

```
GCS::Line &l = Lines[Geoms[geoId].index];

int tag = ++ConstraintsCounter;
GCSsys.addConstraintDifference(l.p1.x, l.p2.x, value,
tag,
     driving);
return ConstraintsCounter;
```

In this example the *addConstraintDifference* function is an example of function of the solver that ultimately implements the creation of a corresponding solver constraint:

```
int System::addConstraintDifference(double *param1,
     double *param2, double *difference, int tagId,
     bool driving)
```

This function allocates dynamic memory for the solver constraint, and adds the parameters given in the constructor to the *pvec* of the constraint. Then stores the tag and driving information therein and relies on a generic solver function *addConstraint* to add the constraint to the system.

```
Constraint *constr = new ConstraintDiffer-
ence(param1,
     param2, difference);
constr->(tagId);
constr->setDriving(driving);
return addConstraint(constr);
```

5.3.3 Constraint adition

This generic solver function resets the previous diagnose results if a new constraint is added, unless it has a negative tag. As previously presented tags are mechanism to map solver constraints to sketcher constraints. However, in addition to that function, negative tags are used to mark solver constraints that do not relate to sketcher constraints, but that are artificially added to achieve a certain functionality. One example is the programatic geometry moving functionality. These constraints, which are only temporarely added do not affect the diagnosis information of the equation system in a way meaningful to the user or the solving process.

The function pushes the new constraint into the vector of solver constraint, and adds the parameters involved in that constraint and the constraint to the adjancency lists of parameters and constraints.

```
isInit = false;
if (constr->getTag() >= 0)
    hasDiagnosis = false;

clist.push_back(constr);
VEC_pD constr_params = constr->params();

for                              (VEC_pD::const_iterator
param=constr_params.begin();
  param != constr_params.end(); ++param) {
    c2p[constr].push_back(*param);
    p2c[*param].push_back(constr);
}
return clist.size()-1;
```

5.4 Initialize the solution and diagnosis

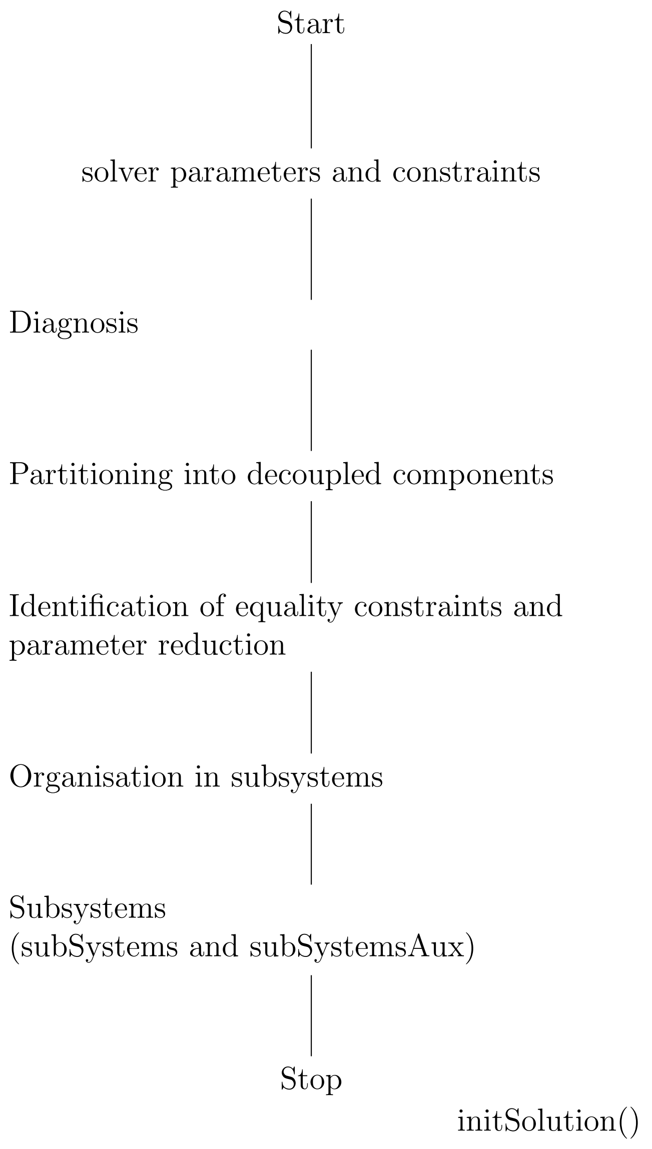
Returning to the sketcher interface function *setUpSketch* in section 4.3, there were some lines of code after the transfer of geometry and constraints to the solver that were not explained in that section.

```
GCSsys.clearByTag(-1);
GCSsys.declareUnknowns(Parameters);
GCSsys.declareDrivenParams(DrivenParameters);
GCSsys.initSolution(defaultSolverRedundant);
```

The first line of code clears any constraint having a priority tag with value -1. After any eventual deletion the code cares so that the constraint to parameter adjacency lists are updated. This effectively removes from the system any constraint that might have been added to programatically move geometry.

The *declareUnknowns* function initializes the *pIndex* map.

Figure 5.1: Flow chart of solution initialisation


Start
solver parameters and constraints
Diagnosis
Partitioning into decoupled components
Identification of equality constraints and
parameter reduction
Organisation in subsystems
Subsystems
(subSystems and subSystemsAux)
Stop
initSolution()

The *declareDrivenParams* function sets the subset of parameters that are datums of driven constraints.

The *initSolution* function is rather complex and performs the following operations:

- Stores the current parameters values in the vector *reference*
- Performs a diagnosis of the whole equation system using a QR decomposition, thereby identifying the degrees of freedom, redundant and conflicting constraints and optionally dependent parameters.
- Identifies any decoupled subsystems and partitions the original system into corresponding components
- Identifies the equality constraints tagged with $ids >= 0$ and prepares a corresponding system reduction
- Organizes the rest of constraints into two subsystems for tag $ids >= 0$ and < 0 respectively

5.4.1 Diagnosis

Diagnosing an equation system as the one of the solver is a rather complex procedure. The function that implements the diagnosis is:

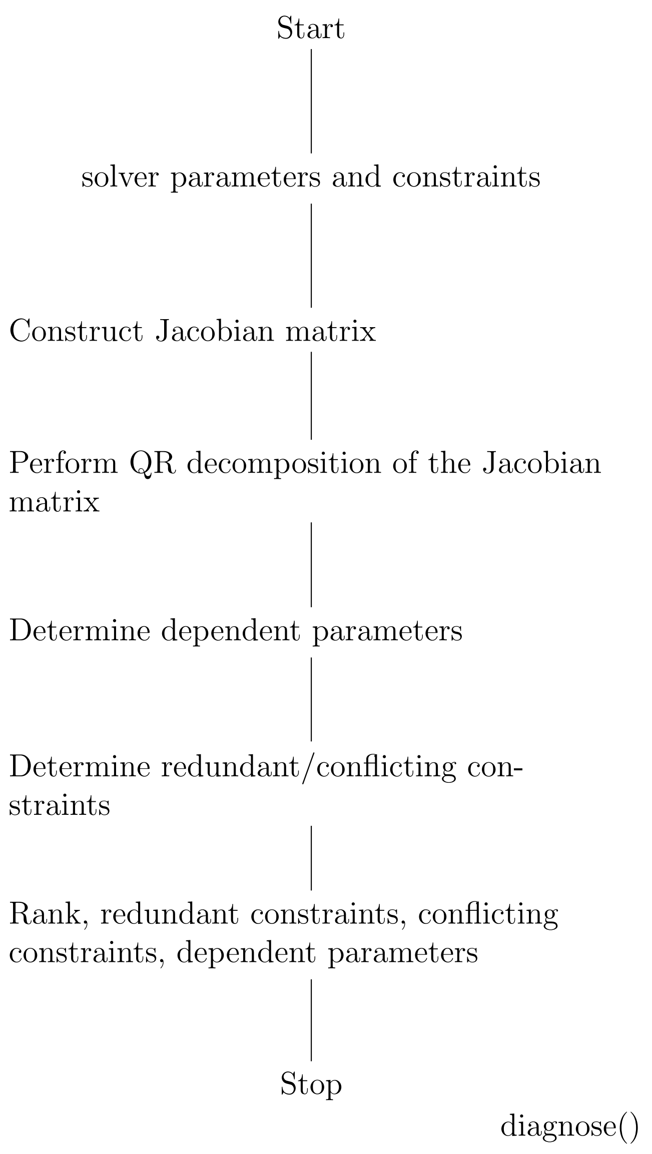
```
int System::diagnose(Algorithm alg)
```

The objective of the diagnosis is to detect redundant and conflicting constraints and calculate the degrees of freedom of the equation system. Optionally, it is possible to also identify parameters that are not fully constrained.

As already anticipate in previous sections, the diagnosis of the equation system is performed using the Jacobian matrix of the system, which is nothing else than a matrix having the gradients of each constraint respect to each of the parameters. Knowing that most constraints only depent on a reduced number of parameters, it is not surprising that such a matrix is substantially sparse (it has a lot of elements that are zero). This is why it is interesting to support sparse algorithms, as they are much more efficient from a computational point of view.

The Jacobian matrix, which is generally a non-square matrix (it has dimensions of the number of constraints times the number of parame-

Figure 5.2: Flow chart of equation system diagnosis


Start
solver parameters and constraints
Construct Jacobian matrix
Perform QR decomposition of the Jacobian matrix
Determine dependent parameters
Determine redundant/conflicting constraints
Rank, redundant constraints, conflicting constraints, dependent parameters
Stop
diagnose()

ters) is decomposed into a QR form. The QR decomposition is a rank-reveling decomposition, meaning that it enables to obtain the rank and thus the degrees of freedom of the equation system. In general, the QR decomposition of the Jacobian matrix J implies finding an orthonormal matrix Q that multiplied by trapezoidal matrix R gives the Jacobian matrix J, so $J = QR$. The trapezoidal matrix R is formed by an upper triangular matrix with zeros at the bottom.

There are different algorithms to obtain a QR decomposition. Different algorithms and variants of such decompositions. Some of the feature *pivoting* or *full pivoting*. Some are dense algorithms, i.e. algorithms designed for dense matrices, while others are sparse algorithms designed for sparse matrices. Unsurprisingly different algorithms provide a different amount of additional information.

In general the QR decomposition involves matrix operations of transposition and permutation. This operations cause that the rows and columns of the Q and R matrices do not correspond to the rows and columns of the J, even though their product sucessfully produces J. Due to the type of matrix, this information is specially important to extract additional information from the trapezoidal matrix R, like the actual constraints that are redundant or conflicting or the parameters that are not fully constraint, i.e. the dependent parameters of the system.

The dense algorithm for QR decomposition is Eigen's *FullPivHouseholderQR*. This decomposition is of the type $PJP' = QR$, where P and P' are permutation matrices. This algorithm enables to obtain information about the dependent parameters of the system.

The sparse algorithm for QR decomposition is Eigen's *SparseQR*. This decomposition is of the type $JP' = QR$, where P' is the column permutation which is the product of the fill-reducing and the rank-revealing permutations.

One historical source of problems in the diagnosis were the reference constraints, which are referred to as driven constraints in the code. Driven constraints create by definition dependent rows, as their datum value is not enforced, while they artificially enlarge the number of parameters and constraints when compared with a case not having such constraints. These problems were addressed by removing the driven constraints from the diagnose. Therefore the diagnose does not operate directly on the parameters vector *plist*, but on a reduced version *pdiagnoselist*.

Removing the reference constraints has the drawback that it makes difficult to identify the solver constraints detected as redundant or conflicting by index. In order to overcome that limitation, a map *jacobianconstraintmap* is created, which assign the index within *plist* to each key having the corresponding index within *pdiagnoselist*.

When redundant or conflict constraints are detected, it is relevant to know whether those redundant or conflicting solver constraints correspond to a single sketcher level constraint, or if the corresponding sketcher level constraint gave rise to a plurality of solver constraints. For this reason, the diagnose function makes use of a *tagmultiplicity* map, which maps the number of solver constraints to a key that is the tag of the constraint.

The Jacobian matrix is created by calculating for each constraint and each parameter, the gradient of the constraint with respect to the parameter:

```
J(jacobianconstraintcount-1,j) =
                              (*constr)-
>grad(pdiagnoselist[j]);
```

Once the Jacobian matrix is created, Eigen is used to calculate the QR decomposition. For reason relating to the availability of permutation information in the different algorithms, a transposed Jacobian matrix is what is actually decomposed. The only effect is that the parameters appear as rows in the decompositions while the constraints appear as columns. The advantage is that column permutation information is available for all the algorithms used.

```
qrJT.compute(
             J.topRows(jacobianconstraintcount).transpose());

paramsNum = qrJT.rows();
constrNum = qrJT.cols();

qrJT.setThreshold(qrpivotThreshold);
rank = qrJT.rank();

if (constrNum >= paramsNum)
   R = qrJT.matrixQR().triangularView<Eigen::Upper>();
else
```

```
R = qrJT.matrixQR().topRows(constrNum)
        .triangularView<Eigen::Upper>();
```

From a practical point of view, it is worth mentioning that in order to determine the rank, the algorithm uses a threshold, which filters out the numbers that while being negligible are not zero. This threshold that is configurable is usually set at around 10^{-13}. Another particularity is that the R matrix in the code is the upper triangular matrix of the trapezoidal matrix.

When using the dense algorithm, Eigen provides the transpositions matrix of the rows. The rows correspond to the parameters, as the QR decomposition was performed on the transposed matrix. With the transposition matrix it is possible to calculate the corresponding permutation matrix of the rows, which can be used to map the parameters corresponding to the rows of the R matrix to the parameters of the J matrix.

The first *rank* rows of the R matrix correspond to the parameters that are fully constrained and form linearly independent columns in the transposed Jacobian matrix. The remaining parameters are the ones that are not fully constrained or dependent parameters, and thus do not contribute to the rank of the system. These dependent parameters are stored in *pdependentparameters*.

Unfortunately, this parameter detection does not seem to work when using the SparseQR algorithm. It may well be that it is not possible or that simply nobody of us have come with the right solution.

The diagnose algorithm continues with the detection of conflicting or redundant constraints. If the number of constraints exceeds the rank of the system, there are necessarily redundant or conflicting constraints.

Just for this paragraph considering that a trapezoidal R matrix, and from the point of view of algebra and taking into consideration that our QR decomposition was performed on transposed Jacobian matrix having N rows corresponding to the N parameters of the system and M columns corresponding to the M constraints, the Q matrix must also have N rows, and the R matrix must also have M columns. Although it is not relevant for the discussion, it is worth noting that the number of colunms of the Q matrix and the number of rows of the R matrix is the same value (otherwise Q could not be right multiplied by R) and that value depends on the rank of the system. The R matrix has a first number of rows that corresponds to the upper triangular matrix and a second number of rows which are zero. The values at the diagonal of this upper triangular matrix will be refered to as *pivots* in the sucessive. By definition in an upper triangular matrix all elements under the pivots, i.e. under the diagonal, are zero. However, the values above the diagonal may take any value. Looking at the first column, it has just one non-zero value, the first one. If we think about how the Q matrix will be right multiplied by the R matrix to produce the transposed jacobian matrix, the values of the first column of it will be determined just on the basis of this non-zero value. It is possible to perform zero reduction operations on the triangular matrix by applying transposition operations. This zero reduction will show that some independent parameters are being defined by more than one constraint, the non-zero values that appear on the upper triangular matrix R after zero reduction, indicating which constraints. This means that either those constraints are conflicting or they are redundant. It is possible to use the column permutation matrix provided by Eigen, in order to map the columns of R with the original constraint from which they have originated. This way it is possible to determine which *solver* constraints are redundant. As more than one non-zero value may actually appear after zero reduction, several *solver* constraints may be conflicting or redundant for a single degree of freedom removed. Therefore, groups of constraints are identified. In the code, this vector of groups is called:

```
std::vector< std::vector<Constraint *>>
                              conflictGroups(constrNum-
rank);
```

The name must seem misleading given the fact that it is not known at this point whether they are redundant or conflicting. In order to detect whether the constraints are redundant or conflicting, the idea is to remove the constraints and try to solve the system, if the error of the constraints is small despite the constraints not having intervened in the solving process, the constraints were *redundant*, whereas if the error is large, they were conflicting. This solution finding is known in the user interface as *redundant solving*, and uses the same algorithms that a normal solving, which will be introduced later on, albeit with differences. These differences will be considered later in the document. It suffices to note that one of the parameters taken by the solving algorithms is just indicating a redundant solving.

There is one main problem remaining before solving. The previous algorithm returns groups of constraints that are conflicting or redundant within the group. However, when two constraints are redundant, a decision must be taken as to which one of them is to be removed. Additionally, a same *solver constraint* may affect several groups. In order to address this problem a heuristic algorithm is executed on these groups until enough constraints have been identified for removal. This identifies for removal the constraints according to these priority rules:

- the constraints that are more popular, i.e. that appear the most among the groups.
- when the popularity is the same, those having a tag with lower multiplicity, i.e. those associated with sketcher constraints that generated a lower number of solver constraints.
- when the popularity and the multiplicity are the same, those with a higher tag number, i.e. those introduced the latest a Sketcher level.

The rationale behind this heuristic algorithm is that if there are several problems, the constraint intervening the most in those problems is the culprit. If there are several such constraints intervening the same amount of times in those problems, it is preferable to get rid of *solver* constraints associated with *sketcher* constraints that generated a lower amount of *solver* constraints. The principle is that the *solver* constraint associated to the less complex *sketcher* constraint should be eliminated, as a *sketcher* constraint may be partly redundant with another *sketcher* constraint, i.e. it may generate a plurality of *solver* constraints, where one of such *solver* constraints is redundant with the *solver* constraints generated from the other *sketcher* constraint, while the others are not. Finally, if the popularity and the multiplicity are the same, the one introduced the latest in the sketcher is chosen. This criteria is the least relevant one, but as one should be chosen anyway, it assumes that the user will have a better change to adapting to the prospective removal of the constraint introduced most recently out of those redundant. It is noted that this paragraph refers to redundancy, but it is still not known whether the constraints are redundant or conflicting.

In order to determine whether the constraints chosen to be removed are redundant or conflicting, an equation system is built without them and a solution is found. If such solution exists, the error of each constraint is checked, and those constraints with an error under the threshold of *convergence* for redundant solving are marked as redundant *solver* constraints.

```
SubSystem *subSysTmp = new SubSystem(clistTmp,
pdiagnoselist);
int res = solve(subSysTmp,true,alg,true);
```

From a programming perspective it is worth mentioning that after solving the solution is applied, the error obtained and after the determination of whether the constraint is redundant or conflicting is made, the parameters are reset to reference values, so that this solving does not affect the solution that will be calculated as described in the following sections.

```
subSysTmp->applySolution();
for (std::set< *>::const_iterator con-
str=skipped.begin();
      constr != skipped.end(); ++constr) {
  double err = (*constr)->error();
  if (err * err < convergenceRedundant)
    redundant.insert(*constr);
}
resetToReference();
```

The redundant constraint set is stored and will be used in other functions within the *initSolution* process.

However, information about the redundant and conflicting constraints is separatedly stored to be retrievable by the *solver interface* and the *sketcher*, in order to inform the user. This information is different from the redundant and conflicting information referred to above in that:

- Conflicting constraints with tag equal to zero are not reported.
- Redundant *solver* constraints associated to a tag comprising other non-redundant *solver* constraints are not reported, i.e. they originate from a *Sketcher* constraint that is partially redundant. The rationale is that there is not point in telling the user that a constraint he cannot remove is partially redundant. There is nothing the user can do about it.

5.4.2 Partitioning into decoupled components

The diagnose process determines redundant constraints, which are removed from the constraint system in order to partion the system into decoupled components. This includes the *solver* constraints that originated from *partially* redundant *sketcher* constraints.

The consequence is that if a *sketcher* constraint is partially redundant, the redundant part is automatically removed. If a *sketcher* constraint is fully redundant, it is reported. It has been asked several times why not have the solver ignore the redudant constraints altogether. The answer is that eliminating *sketcher* constraints helps prevent more complex problems and helps the user have cleaner sketches.

The decoupling into components is effected using a graph. First a vertex is added per parameter and per constraint. Then, for each constraint, the adjacent parameters are iterated and an edge is created connecting the corresponding constraint vertex with the index the index in the parameters vector of the given parameter.

This enables to determine how many components, unconnected vertices, are present.

5.4.3 Identification of equality constraints and parameter reduction

Parameter reduction is effected by eliminating constraints by reduction. The eliminated constraints are stored in:

```
std::set<Constraint *> reducedConstrs;
```

These parameter reductions are calculated and stored in a vector of maps called *reductionmaps*, storing a reductionmap per component, i.e. decoupled subsystem.

```
std::vector< std::map < double * , double *>> reduc-
tionmaps;
```

The reduction map is calculated by iterating the *solver* constraints and for those that are equality constraints, the first parameter is kept, while the second parameter is reduced. This means that the second parameter is made to point to the first parameter. For each of the components this association of reduced parameters is stored in the map.

The parameters and constraints after reduction are stored in *clists* and *plists* respectively. This includes, per component, all the constraints that are not the reduced ones.

5.4.4 Organisation in subsystems

This step uses the information of parameters and constraints after reduction together with the *reductionmaps* in order to organise the resulting parameters and constraints into subsystems.

As explained in previous sections, the actual subsystems to be solved are stored in vectors of *GCS::SubSystem* type.

```
std::vector<SubSystem *> subSystems, subSystem-
sAux;
```

These vectors are first cleared and any previously allocated memory released. Then the *reduced* constraints per decoupled component are iterated. Two subsystems per component are created, one stored into *subSystems* and the other stored into *subSystemsAux*. The former containts the subsystems formed by the *reduced* constraints with non-negative tags, whereas the latter contains the subsystems formed by the *reduced* constraints with negative tags. As explained before the constraints with negative tag originate from programatic movements of geometry.

5.5 Solve

There is one *general* solving function at *solver* level. There are three *specific* solving functions intended for special circumnstances and to be used by the *general* solving function. There are four *algorithms* that implement modified newton methods to perform the actual solving.

5.5.1 Parameter solve

The first *specific* solving function takes the *parameters* to solve. It initializes a solution for those parameters and solves the system using the *general* solving function. Hence, the specific part is the particular initilization directly from *parameters*. This function is currently *not* used by FreeCAD.

```
int System::solve(VEC_pD &params,
                  bool isFine,
                  Algorithm alg,
                  bool isRedundantsolving)
```

5.5.2 Single subsystem solve

The second *specific* solving function takes one subsystem with no priority constraints, i.e. without any constraint with negative tags. This function is also the function used for the *redundant* solvings to determine whether a constraint is *redundant* or *conflicting* during the *diagnose* (see section 5.4). The actual function is:

```
int System::solve(SubSystem *subsys,
                  bool isFine,
                  Algorithm alg,
                  bool isRedundantsolving)
```

Solving a single subsystem is the most frequent solving operation and just relies on the actual implementation of the solving *algorithms* implemented for a single subsystem solving. Currently, those algorithms are:

- *Broyden-Fletcher-Goldfarb-Shanno*, BFGS in the following.
- *Levenberg-Marquardt*, LM in the following.
- *DogLeg*, DL in the following.

The functions implementing those *algorithms* are:

```
int System::solve_BFGS(SubSystem *subsys,
                  bool /*isFine*/,
                  bool isRedundantsolving)

int System::solve_LM(SubSystem* subsys,
                  bool isRedundantsolving)

int System::solve_DL(SubSystem* subsys,
                      bool isRedundantsolving)
```

5.5.3 Two subsystem solving

The third *specific* solving function is actually the implementation of an *algorithm*. This is because there is only one *algorithm* for the simultaneous solving of two subsystems. The algorithm gives priority to one subsystem over the other. This is the algorithm used for *programatic* moving operations like dragging. The function is:

```
int System::solve(SubSystem *subsysA,
                      SubSystem *subsysB,
                      bool /*isFine*/,
                      bool isRedundantsolving)
```

5.5.4 General solve

The function that is called by the *solver interface* in order to effect a solve operation as seen in section 4.4 is:

```
int System::solve(bool isFine,
                      Algorithm alg,
                      bool isRedundantsolving)
```

A solve operation at *solver* level only proceeds if the system is initialized, i.e. the subsystem(s) to be solved are created.

The different decoupled components, i.e. decoupled subsystems are solved separatedly. For each decoupled component, three possibilities may arise:

- That only a *subSystems* exists.
- That only a *subSystemsAux* exists.
- That both a *subSystems* and a *subSystemsAux* exist.

For each case in each decoupled component, one of the following solving functions is executed:

```
int System::solve(SubSystem *subsys,
                  bool isFine,
                  Algorithm alg,
                  bool isRedundantsolving)

int System::solve(SubSystem *subsysA,
                  SubSystem *subsysB,
                  bool /*isFine*/,
                  bool isRedundantsolving)
```

After all the components have been solved, if all the solvings returned *GCS::Success*, all the constraints are iterated and the *error* is checked. If the error of any individual constraint is greater than the *convergence* configured value, then the result is changed to *GCS::Converged*, indicating that the solver did indeed converge, but the convergence did not lead to the constraints having been enforced at the required *error* level.

As it is apparent, when two systems are provided, the *two subsystem solving* is executed using the priority solving algorithm, whereas when one system is provided, the *single subsystem solving* is executed using the *algorithm* provided as a parameter. It is noted that when the subsystem contains both constraints with positive and negative tags, it does not matter which value the *Algorithm* parameter takes, the specific two system's algorithm is executed. This parameter is only taken into account when a single subsystem is provided.