Numerical Methods for Digital Geometry Processing extended abstract

Bruno Lévy
ALICE

INRIA
54600, Villers-les-Nancy, France
levy@loria.fr

Abstract

Digital Geometry Processing recently appeared (in the middle of the 90's) as a promising avenue to solve the geometric modeling problems encountered when manipulating surfaces represented by discrete elements (i.e. meshes). Since a mesh may be considered to be a *sampling* of a surface - in other words a *signal* - the DGP (*digital signal processing*) formalism was a natural theoretic background for this discipline (see e.g. [20]). In this discipline, discrete fairing [13] and mesh parameterization [9] have been two active research topics these last few years.

In parallel with the evolution of this discipline, acquisition techniques have made huge advances, and todays meshes acquired from real objects by range-laser scanners are larger and larger (30 million triangles is now common). This causes difficulties when trying to apply DGP tools to these meshes. The kernel of a DGP algorithm is a numerical method, used either to solve a linear system, or to minimize a multivariate function. The Gauss-Seidel iteration and gradient descent methods used at the early ages of DGP do not scale-up when applied to huge meshes.

In this presentation, our goal is to give a survey of classic and more recent numerical methods, to show how they can be applied to DGP problems, from a theoretic point of view down to implementation. We will focus on two different classes of DGP problems (mesh fairing and mesh parameterization), show solutions for linear problems, quadratic problems, and general non-linear problems, with and without constraint. In particular, we give a general formulation of quadratic problems with reduced degrees of freedom that can be used as a general framework to solve a wide class of DGP problems. Our method is implemented in the OpenNL library, freely available on the web. The presentation will be illustrated with live demos of the methods.

Keywords: DGP, numerical optimization

1 Introduction

Since the seminal work by Gabriel Taubin [20] in Digital Geometry Processing, 3D acquisition and processing methods have made huge advances. As a consequence, the volume of the data manipulated by these algorithms has grown by several orders of magnitude. For this reason, it is necessary to replace the Gauss-Seidel iteration and gradient descent methods used at the early ages by more sophisticated methods. In this paper, we give an overview of efficient numerical methods. In addition, we introduce a specific formulation of quadratic problems with reduced degrees of freedom, that can be used to implement a general framework. This framework can be used to solve a wide class of DGP problems. We have implemented this idea in the "OpenNL" library, freely available from our web site[5], and interfaced with existing efficient solvers (SuperLU[6], MUMPS[2] and TAUCS[21]).

Before diving into the heart of the matter, we quickly introduce below two important aspects of Digital Geometry Processing, namely mesh parameterization and discrete fairing, together with some classic numerical problems encountered in this area. We will then show different methods to solve those numerical problems.

1.1 Mesh parameterization

Mesh parameterization is a problem for which the Digital Geometry Processing have been investing much activity these last few years. "Curves and Surfaces" geometric models are represented by parametric functions. This representation is useful for many application domains, including attaching properties to the surface (they can be represented by 2D data structures in parameter-space), or meshing algorithms. For this reason, methods to obtain a parametric representation from a mesh model were investigated. In his pioneering work[9], motivated by a Spline fitting problem, Michael Floater had the idea to use Tutte's barycentric mapping theorem[22] to construct a piecewise linear parameterization of a triangulated mesh homeomorphic to a disc. Tutte's barycentric theorem states that given a triangulation, given 2D coordinates $\mathbf{u}_i = (u_i, v_i)$ associated with each vertex i of the triangulation, the following two conditions are sufficient to ensure that the $\mathbf{u}_i$'s define a valid (i.e. non-overlapping) parameterization:

(1) the boundary is a convex polygon

(2) for all interior vertex i, $d_i \mathbf{u}_i = \sum_{j \in N_i} \mathbf{u}_j$

where d_i denotes the degree (or valence) of vertex i and where N_i denotes the set of vertices connected to vertex i by an edge.

Floater's approach is based on the remark that besides characterizing a class of valid parameterizations, Tutte's theorem gives a way of constructing a parameterization. One just needs to distribute the boundary vertices on a convex polygon, and solve for the (u_i, v_i)'s of the interior vertices in condition (2). In practice, this means solving two linear systems, one for the (u_i)'s and the other one for the (v_i)'s. We will review in the next section different methods to achieve this.

Many papers were then published on the specific topic of mesh parameterization, relaxing the constraint of using a fixed convex boundary in parameter space, and minimizing different deformation criteria, adapted to different application domains. The recent survey[10] lists the most significant advances in this area. A large category of these methods ends up with minimizing a quadratic function. For instance, the discretization of *harmonic maps* proposed in [8] means minimizing the following energy functional:

$$F_{harmonic} = \sum_{(i,j)\in E} a_{i,j} \|\mathbf{u}_i - \mathbf{u}_j\|^2 \qquad (1)$$

where the $a_{i,j}$'s are coefficients that depend on the geometry of the surface. If the $a_{i,j}$'s are defined by using the famous *cotangent weights* proposed by Pinkall and Polthier in [17], this defines a discrete harmonic energy. This was later refined by Desbrun et. al in [7], where free boundary conditions were introduced, together with a geometric interpretation of the gradients.

We simultaneously developed an equivalent method in [14], taking the dual path of minimizing the conformal energy of the parameterization, defined by:

$$F_{conformal} = \sum_{T} A_T \|\nabla u - \mathrm{rot}_{90}(\nabla v)\|^2 \qquad (2)$$

where T denotes a triangle, A_T its area in 3D space, and rot_{90} a 90 degrees rotation. Both approaches result in a quadratic objective function to minimize. We will review and compare different methods in the next section.

1.2 Discrete fairing

Adapting to meshed models all the modeling tools available with the "Curves and Surfaces" representation is another challenge of the Digital Geometry Processing discipline. In "Curves and Surfaces" representations, the geometry is represented by a set of parametric surfaces. Time and effort has been devoted to the problem of optimizing the shape of a surface, by minimizing a "fairness" criterion. Fairness is often defined using notions from differential geometry (mean curvature, Gaussian curvature ...) or approximation of physics (thin-plate energy). In general, optimizing the fairness means solving a Partial Differential Equation [3]. Adapting this formalism to the case of a discrete mesh model was an active research area. Kobbelt coined the term *discrete fairing* in [13] to qualify this family of approaches. In the context of this paper, we will use as an example the formulation given in [16] of the Discrete Fairing problem:

$$F_{smooth} = \sum_{i} \|d_i \mathbf{p}_i - \sum_{j\in N_i} \mathbf{p}_j\|^2 \qquad (3)$$

where $\mathbf{p}_i = (x_i, y_i, z_i)$ denote the coordinates at the vertices of the triangulation and d_i the degree of vertex i.

As in the previous subsection, the energy functional F_{smooth} is a quadratic function of the variables. In the next section, we will review different numerical algorithms to minimize this energy functional.

2 Numerical methods

In all the DGP methods listed in the section above (Floater's parameterization method, harmonic parameterization, conformal parameterization and discrete fairing), we need either to solve a linear system or to minimize a quadratic objective function. A specificity of the numerical problems yielded in DGP is that the involved matrices are usually sparse. As a consequence, we will first explain how to efficiently implement a data structure for sparse matrices. To make user's life easier, we will propose an implementation that can dynamically grow when coefficients are added to the matrix. A C++ version of this implementation is given in Appendix A, based on the `std::vector` data structure. This data structure is available in our Graphite software. We also provide a C version in OpenNL[5]. Note that this data structure can be easily extended. Our C++ and C implementations also provide the following features (not detailed in the implementation given in the appendix to keep its length reasonable):

1. storage of the diagonal term
2. storage of both sparse rows and columns
3. symmetric storage (i.e., do not store the upper triangle for symmetric matrices)
4. non-square matrices
5. matrix $\times$ matrix multiply

Features (1) and (2) are interresting for implementing Jacobi preconditioner (requires 1) or SSOR preconditioner (requires 1+2). Feature (3) speeds up the conjugate gradient algorithm (but requires some modification in the matrix $\times$ vector routine). Features (4) and (5) can be used by more sophisticated non-linear solvers (such as our ABF++ algorithm [19]).

2.1 Linear systems

We will review different methods, including the classic Gauss-Seidel optimization, the Conjugate Gradient algorithm, preconditioners (see e.g., [1]) and sophisticated sparse direct methods, such as SuperLU[6], MUMPS[2] and TAUCS[21]. We will show how to implement and/or use these methods, and how they behave for problems ranging from several thousand variables to millions of variables. Basically, implementing an iterative solver (like the Conjugate Gradient algorithm) only requires efficient matrix-vector multiplies. An implementation is proposed in Appendix A. If we want to benefit from both the flexibility of our dynamically growing sparse matrix data structure and the efficiency of a sparse direct solver (MUMPS, TAUCS), it is necessary to convert the dynamic sparse matrix into the more standard CCS representation (compressed column storage).

The CCS representation cannot handle dynamically growing matrices. As in our case, it is based on storing only the non-zero entries of the matrix, in an array `a` of dimension `nnz` (number of non-zero coefficients). To represent the sparsity pattern of the matrix, this array is complemented with two arrays of indices. In a nutshell, the CCS representation is as follows:

- `a` (of size nnz), coefficients of the matrix
- `row_ind` (of size nnz): `row_ind[i]` corresponds to the row index of `a[i]`
- `col_ptr` (of size n+1): `col_ptr[i]` indicates the index in `a` from which the coefficients of column `i` are stored. By convention, `col_ptr[n] = nnz`

The conversion algorithm is trivial (see Appendix B), and for a reasonably large system, it requires only a negligible time as compared to the time spent in the solver. Using the dynamic sparse matrix data structure (Appendix A), it is easy to dynamically construct a linear system from a DGP equation and a mesh, by traversing the mesh and adding the terms to the matrix. Using the conversion routine (Appendix B), our dynamic sparse matrix data structure can be interfaced with a large number of existing solvers. Note: when interfacing with FORTRAN and other third-party routines, one needs to take care about the array indexing conversion ($0 \dots n-1$ in C and $1 \dots n$ in Fortran) and setting the `array_base` parameter in consequence. The indexing convention is most of the time indicated in the documentation of the routine.

2.2 Quadratic minimization

In this section, we consider the problem of minimizing a quadratic form F given by

$$F(x) = \|Ax - b\|^2$$

where x is the vector of unknowns (of dimension n), A is a $m \times n$ matrix (m is usually larger than n), and b is a vector of dimension m. Note that in DGP, we commonly need to fix some variables. For instance, in discrete fairing, it is common to consider that some of the vertices are fixed, and that the other vertices are free to move. For this reason, we consider the possibility of removing degrees of freedom from F. Formally, this means that the vector x of unknowns is decomposed into two sub-vectors, x_f and x_l, where x_f denotes the set of variables that are $f$ree to move, and where x_l denotes the set of $l$ocked variables. The energy functional then becomes (in block matrix notation):

$$F(x_f) = \left\| [A_f | A_l] \begin{bmatrix} x_f \\ \hline x_l \end{bmatrix} - b \right\|^2$$

where A is split into A_f and A_l according to x_f and x_l. Note that the energy functionals minimized in harmonic parameterization (Equation 1), conformal parameterization (Equation 2) and discrete fairing (Equation 3) are a specific instance of this equation. The minimizer of $F(x_f)$ satisfies $\nabla F(x_f) = 0$, where $\nabla F(x_f) = 2A_f^t A_f x_f - 2A_f^t b + 2A_f^t A_l x_l$. In other words, finding the minimizer x_f of $F(x_f)$ means solving the following linear system:

$$M x_f = c$$

$$\text{where: } \begin{cases} M &= A_f^t A_f \\ c &= A_f^t b - A_f^t A_l x_l \end{cases} \quad (4)$$

As a consequence, we are again faced with a linear system to solve. A particularity is that the matrix $A_f^t A_f$ is symmetric, which enables us to use methods optimized for symmetric matrices (conjugate gradient and sparse cholesky factorization). We will show some results and statistics of those methods applied to parameterization and mesh fairing problems.

Note that in both harmonic parameterization, conformal parameterization and discrete fairing, it is easier to construct the matrix A and the right-hand side b in a row-by-row order:

- In harmonic parameterization and discrete fairing, each vertex yields one row in A and b.
- In conformal parameterization, each triangle yields one row in A and b.

Generally, all DGP methods minimize an expression, involving terms attached to all the vertices (or edges, or triangles) of the mesh. These terms (also called *stencils* in the finite elements community) involve small neighborhoods of the vertices, edges or triangles. Therefore, constructing the expressions involved in the numerical optimization process means traversing all the vertices (or edges, or triangles) of the mesh. This gives the coefficients of the the matrix A and the right hand side b in row-major order. Based on this remark, to facilitate the implementation of these methods, we show how to incrementally compute the terms of the problem with reduced degrees of freedom (Equation 4).

From a practical point of view, this gives a unified framework to implement various parameterization and mesh fairing methods that we will demonstrate. These concepts are implemented in our "OpenNL" library, with a syntax similar to the OpenGL graphics library: the lines of A and b are simply constructed by calling sequences of functions `nlBegin()`,`nlAddCoefficient(i,a)`,...,`nlEnd()`. Our algorithm automatically updates the matrix $M = A_f^t A_f$ and the right hand side $c = A_f^t b - A_f^t A_l x_l$ for each row k of A and b. They are updated by applying the following scheme (simply yielded by ordering the terms of the linear system by the index k):

for i **from** 1 **to** nf
  for j **from** 1 **to** nf
    $m_{i,j} \leftarrow m_{i,j} + a_{k,i} \times a_{k,j}$
  end for
end for
$S \leftarrow -b_k$
for j **from** $nf+1$ **to** $nf+nl$
  $S \leftarrow S + a_{k,j} \times x_j$
end for
for i **from** 1 **to** nf
  $c_i \leftarrow c_i - a_{k,i} \times S$
end for

Note that the updating formula depends on the locked variables $x_{nf+1} \dots x_{nf+nl}$. In practice, to reduce computations, we only store the non-zero entries of the row $a_{k,.}$ (together with the corresponding indices). This formulation, combined with our dynamically growing sparse data structure (Appendix A), is especially convenient to construct the type of linear system mentioned above. Our OpenNL library is freely available on the web [5]. OpenNL was successfully integrated in the Blender 3D modeler, and used to implement texture atlas generation tools based on our LSCM method. We will show in the presentation how to easily implement various parameterization and fairing algorithms based on this framework.

2.3 Non-linear minimization

However, some numerical methods do not fall into the two categories above (i.e. linear systems and quadratic energy functionals). For instance, Hormann et. al's MIPS parameterization method [12] is based on a non-linear energy functional. The energy minimized by the ABF method [18] is quadratic, but the introduction of constraints required by the method transforms it into a non-linear function. More recent works, such as the Discrete Willmore Flow [4] also involve the minimization of a non-linear function.

A possible strategy is to use a simple gradient descent method, and apply it in a multi-resolution setting. This was successfully applied to MIPS by Hormann. However, it is difficult to tune the different parameters involved in the process (i.e. metric used by the mutliresolution algorithm, number of levels, number of iterations per level ...). Another possible alternative is to use Newton's method, defined as follows:

$$\begin{array}{l} \textbf{while} \quad \|\nabla F(x)\| > \epsilon \\ \quad \textbf{solve} \quad \nabla^2 F(x)\delta = -\nabla F(x) \\ \quad x \leftarrow x + \delta \\ \textbf{end} \end{array} \tag{5}$$

As can be seen, this means solving a series of linear systems. We will show in the presentation some examples of Newton's method applied to simple and more complicated cases. We will show how we successfully applied this method in [19] together with several optimization techniques (Schur complement) to parameterize meshes of millions vertices.

Conclusion and Perspectives

We have given a quick overview of numerical methods, from an insider's perspective instead of considering them as black boxes. The sparse matrix implementation given in Appendix A is fully functional (with only 58 lines of source code), and can solve problems with thousands of unknowns (by adding the corresponding conjugate gradient routine, i.e. a handfull of C++ lines). To solve larger problems (up to million variables), interested readers can download our freely available implementation. In addition, our formulation of least squares problem facilitates the implementation and experimentation of new DGP algorithms.

With this project, one of our goals was to experiment with the *futurist programming* philosophy [11] and produce a programming library useful to a wider community than the DGP research community. The result of this experiment is OpenNL [5], a minimalist numerical library. In less than 6000 lines of portable C, it is possible to efficiently solve large sparse linear systems and minimize quadratic forms with reduced degrees of freedom. The *usability* goal was reached, since OpenNL was integrated into the widely used 3D modeler *Blender* in less than two weeks (by Brecht Van Lommel and Jens Ole Wund).

We will conclude the talk by giving some perspectives about future works. More specifically, we will mention a new class of functional optimization problems where both the coefficients and function bases are unknowns. We will also mention research in mixed symbolic/numeric solvers, and show early results in this area.

Acknowledgments

Thanks to the EU Network of Excellence `AIM@Shape` (IST NoE No 506766) and to the ARC GEOREP (INRIA grant) for supporting this work. Thanks to Brecht Van Lommel and Jens Ole Wund for their fantastic work in Blender (they implemented LSCM unwrapping based on OpenNL, see [15]).

A Sparse Matrix Data Structure

```
class SparseMatrix {
public:

    struct Coeff {
        Coeff() { }
        Coeff(unsigned int i, double val) : index(i), a(val) { }
        unsigned int index ;
        double a ;
    } ;

    class Column : public std::vector<Coeff> {
    public:
        void add(unsigned int index, double val) {
            for(unsigned int i=0; i<size(); i++) {
                if((*this)[i].index == index) {
                    (*this)[i].a += val ;
                    return ;
                }
            }
            std::vector<Coeff>::push_back(Coeff(index, val)) ;
        }
    } ;

    SparseMatrix(unsigned int dim) : dimension(dim) {
        column = new Column[dim] ;
    }

    ~SparseMatrix() { delete[] column;  }

    // aij <- aij + val
    void add(unsigned int i, unsigned int j, double val) {
        column[j].add(i, val) ;
    }

    // A <- 0
    void clear() {
        for(unsigned int j=0; j<dimension; j++) {
            column[j].clear() ;
        }
    }

    // y <- Ax
    void mul(double* x, double * y) {
        for(unsigned int i=0; i<dimension; i++) {
           y[i] = 0 ;
        }
        for(unsigned int j=0; j<dimension; j++) {
            const Column& C = column[j] ;
            for(unsigned int i=0; i<column[j].size(); i++) {
                y[ C[i].index ] += C[i].a * x[j]  ;
            }
        }
    }

   // number of non-zero coefficients
   unsigned int nnz() const {
      unsigned int result = 0 ;
      for(unsigned int i=0; i<dimension; i++) {
          result += column[i].size() ;
      }
      return result ;
   }

    unsigned int dimension ;
    Column* column ;
} ;
```

B Converting to CCS format

```
void to_CCS(
   const SparseMatrix& M,
   double*& a, unsigned int*& row_ind, unsigned int*& col_ptr,
   unsigned int array_base = 0
) {
   unsigned int n = M.dimension ;
   unsigned int nnz = M.nnz() ;
   a = new double[nnz] ;
   row_ind = new unsigned int[nnz] ;
   col_ptr = new unsigned int[n+1] ;
   unsigned int count = 0 ;
   for(unsigned int j=0; j<n; j++) {
        const SparseMatrix::Column& C = M.column[j] ;
        row_ptr[j] = count + array_base ;
        for(unsigned int i=0; i<C.size(); i++) {
            a[count] = C[i].a ;
            row_ind[count] = C[i].index + array_base ;
            count++ ;
        }
    }
    col_ptr[n] = nnz + array_base ;
}
```

REFERENCES

[1] Burak Aksoylu, Andrei Khodakovsky, and Peter Schröder. Multilevel solvers for unstructured surface meshes. *SIAM J. Sci. Comput, in review*, 2005.

[2] P. Amestov, A. Guermouche, J.Y. L'Excellent, and MUMPS S. Pralet, 2005. http://graal.ens-lyon.fr/MUMPS/.

[3] M.I.G. Bloor and M.J. Wilson. Using PDEs to generate free-form surfaces. *CAD*, 22, 1990.

[4] A. Bobenko and P. Schroeder. Discrete willmore flow. In *SGP conf. proc.*, 2005.

[5] OpenNL Bruno Levy, 2004. http://www.loria.fr/~levy/software/index.html.

[6] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.

[7] Mathieu Desbrun, Mark Meyer, and Pierre Alliez. Intrinsic parameterizations of surface meshes. In *Proceedings of Eurographics*, pages 209–218, 2002.

[8] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. In *SIGGRAPH Conference Proceedings*, pages 173–182. ACM, 1995.

[9] M. Floater. Parametrization and smooth approximation of surface triangulations. *Computer Aided Geometric Design*, 14(3):231–250, April 1997.

[10] M. S. Floater and K. Hormann. Surface parameterization: a tutorial and survey. In M. S. Floater N. Dodgson and M. Sabin, editors, *Advances on Multiresolution in Geometric Modelling*. Springer-Verlag, 2004.

[11] Paul Haeberli. Futurist programming notes. http://www.sgi.com/misc/grafica/future/futnotes.html.

[12] K. Hormann and G. Greiner. MIPS. In *Curve and Surface Design*, pages 153–162. Vanderbilt University Press, 2000.

[13] L. Kobbelt. Discrete fairing. In *Proceedings of the Seventh IMA Conference on the Mathematics of Surfaces*, pages 101–131, 1997.

[14] Bruno Lévy, Sylvain Petitjean, Nicolas Ray, and Jérome Maillot. Lscm. *ACM Transactions on Graphics (SIGGRAPH conf. proc.)*, pages 362–371, 2002.

[15] B. V. Lommel and J. O. Wund. Uv texture coordinate using lscm. http://blender3d.org/cms/UV_Unwrapping.363.0.html.

[16] J.L. Mallet. Discrete Smooth Interpolation. *Computer Aided Design*, 24(4):263–270, 1992.

[17] U. Pinkall and K. Polthier. Computing discrete minimal surfaces and their conjugates. *Experimental Math.*, 2(15), 1993.

[18] Alla Sheffer and Eric de Sturler. Parameterization of faceted surfaces for meshing using angle based flattening. *Engineering with Computers*, 17:326–337, 2001.

[19] Alla Sheffer, Bruno Levy, Maxim Mogilnitsky, and Alexander Bogomyakov. Abf++. *ACM TOG*, Apr 2004.

[20] G. Taubin. A signal processing approach to fair surface design. In *SIGGRAPH Conference Proceedings*, pages 351–358. ACM, 1995.

[21] S. Toledo, D. Chen, and TAUCS V. Rotkin, 2003. http://www.tau.ac.il/~stoledo/taucs/.

[22] W. Tutte. Convex representation of graphs. In *Proc. London Math. Soc.*, volume 10, 1960.