Methods for Smoothing Experimental Data in the smooth Program

Technical Documentation
Version 5.1 | May 2025

Contents

Introduction

The `smooth` program implements three sophisticated methods for smoothing experimental data with the capability of simultaneous derivative computation. Each method has specific properties, advantages, and areas of application.

General Smoothing Problem

Experimental data often contains random noise:

`y_obs(x_i) = y_true(x_i) +  _i`

The goal of smoothing is to estimate `y_true` while suppressing ` _i` and preserving physically relevant signal properties.

Program Structure

`./smooth [options] data_file`

Basic Parameters: - `-m {0|1|2}` - method selection (polyfit|savgol|tikhonov) - `-n N` - smoothing window size (polyfit, savgol) - `-p P` - polynomial degree (polyfit, savgol, max 12) - `-l ` - regularization parameter (tikhonov) - `-l auto` - automatic selection using GCV (tikhonov) - `-a` - use adaptive weights for non-uniform grids (tikhonov) - `-d` - display first derivative in output (optional)

Note on polynomial degree: Degrees > 6 may generate numerical stability warnings.

Note on derivatives: From version 5.1, first derivative output is optional. Without the `-d` switch, the program outputs only smoothed values. With the `-d` switch, it outputs both smoothed values and first derivatives.

API Structure

All smoothing methods have consistent APIs:

```c
// Polyfit
PolyfitResult* polyfit_smooth(double *x, double *y, int n,
                              int window_size, int poly_degree);
void free_polyfit_result(PolyfitResult *result);

// Savitzky-Golay
SavgolResult* savgol_smooth(double *x, double *y, int n,
                            int window_size, int poly_degree);
void free_savgol_result(SavgolResult *result);

// Tikhonov
TikhonovResult* tikhonov_smooth(double *x, double *y, int n, double lambda);
TikhonovResult* tikhonov_smooth_adaptive(double *x, double *y, int n,
                                         double lambda, int adaptive_weights);
void free_tikhonov_result(TikhonovResult *result);
```

Polynomial Fitting (POLYFIT)

Mathematical Foundations

The POLYFIT method uses local polynomial fitting with least squares method in a sliding window.

Problem: For each point `x_i`, we fit a polynomial of degree `p` to the surrounding `n` points:

```
P(x) = a_0 + a_1(x-x_i) + a_2(x-x_i)² + ... + a_p(x-x_i)^p
```

Optimization criterion:

```
min Σ[y_j - P(x_j)]²    for j   [i-n/2, i+n/2]
```

Construction of Normal Equations

For polynomial coefficients, we solve a system of linear equations:

```
[Σ(x-x_i)     Σ(x-x_i)¹     ...  Σ(x-x_i)^p  ] [a_0]   [Σy(x-x_i) ]
[Σ(x-x_i)¹     Σ(x-x_i)²     ...  Σ(x-x_i)^{p+1}] [a_1] = [Σy(x-x_i)¹]
[                                            ] [   ]   [          ]
[Σ(x-x_i)^p Σ(x-x_i)^{p+1} ... Σ(x-x_i)^{2p}] [a_p]   [Σy(x-x_i)^p]
```

where summation is over points in the window around `x_i`.

Derivative Computation

Derivatives are computed analytically from polynomial coefficients:

```
f(x_i) = a_0
f'(x_i) = a_1
f''(x_i) = 2a_2
```

Edge Handling

At edges, asymmetric windows are used with extrapolation of the fitted polynomial:

```
// For point x_k < x_{n/2}
f(x_k) = Σ_{m=0}^p a_m * (x_k - x_{n/2})^m
```

Efficient Implementation

The program uses LAPACK routine `dposv` for solving symmetric positive definite systems at each point:

```
// System solution for polynomial coefficients
dposv_(&uplo, &matrix_size, &nrhs, C, &matrix_size, B, &matrix_size, &info);
```

The normal equations matrix is symmetric and positive definite, making `dposv` optimal for this application.

Modularized Implementation

```
// polyfit.h
typedef struct {
    double *y_smooth;     // Smoothed values
    double *y_deriv;      // First derivatives
    int n;                // Number of points
    int poly_degree;      // Polynomial degree
    int window_size;      // Window size
} PolyfitResult;
```

Characteristics

Advantages: - Excellent local approximation - Analytical computation of derivatives of any order - Adaptable to changes in curvature - Good preservation of local extrema

Disadvantages: - Sensitive to outliers - Boundary effects at edges - Possible Runge oscillations for high polynomial degrees (p > 6) - Numerical instability warnings for degrees > 6

Savitzky-Golay Filter (SAVGOL)

Theoretical Foundations

The Savitzky-Golay filter is an optimal linear filter for smoothing and derivatives based on local polynomial regression. The key innovation is pre-computation of convolution coefficients.

Fundamental principle: For given parameters (window size, polynomial degree, derivative order), there exist universal coefficients `c_k` such that:

```
f^(d)(x_i) = Σ_{k=-n_L}^{n_R} c_k · y_{i+k}
```

Key Difference from POLYFIT Method

While both SAVGOL and POLYFIT use polynomial approximation, they differ fundamentally in their computational approach:

POLYFIT approach: - For each data point, fits a new polynomial to the surrounding window - Solves the least squares problem individually for each point - Coefficients of the polynomial change with each window position - Computationally intensive: $O(n \cdot p^3)$

SAVGOL approach (Method of Undetermined Coefficients): - Recognizes that for equidistant grids, the filter coefficients are translation-invariant - Uses the **method of undetermined coefficients** to pre-compute universal weights - These weights depend only on the window geometry, not on the actual data values - Applies the same weights as a linear convolution across all data points - Computationally efficient: $O(p^3)$ once, then $O(n \cdot w)$ for application

The Method of Undetermined Coefficients

The Savitzky-Golay method seeks a linear combination of data points:

```
ŷ  = c  ·y   + ... + c ·y  + ... + c  ·y
```

where the coefficients `c_k` are "undetermined" and must satisfy the condition that the filter exactly reproduces polynomials up to degree `p`.

The key insight: For a given window configuration and polynomial degree, these coefficients can be determined once and applied universally.

Coefficient Derivation

Coefficients are derived from the condition that the filter must exactly reproduce polynomials up to degree `p`.

Moment conditions:

```
Σ_{j=-n_L}^{n_R} c_j · j^m =  _{m,d} · d!    for m = 0,1,...,p
```

where: - ` _{m,d}` is the Kronecker delta - `d` is the derivative order - `d!` is factorial

This leads to a system of linear equations where the unknowns are the filter coefficients `c_j`.

Matrix Formulation

We solve a system of linear equations:

```
[1   -n_L    (-n_L)²    ...  (-n_L)^p  ] [c_{-n_L}]    [ _{0,d}·0!]
[1  -n_L+1  (-n_L+1)²  ... (-n_L+1)^p ] [c_{-n_L+1}] = [ _{1,d}·1!]
[                                      ] [         ]   [          ]
[1    n_R     n_R²      ...    n_R^p   ] [  c_{n_R} ]   [ _{p,d}·p!]
```

Computational Efficiency

The brilliance of the Savitzky-Golay approach becomes apparent when processing large datasets:

Example for 10,000 data points, window size 21, polynomial degree 4: - POLYFIT: Must solve 10,000 separate 5×5 linear systems - **SAVGOL:** Solves only ONE 5×5 system, then performs 10,000 simple weighted sums

This difference explains why SAVGOL is preferred for real-time signal processing and large datasets, while maintaining the same mathematical accuracy as POLYFIT for uniform grids.

Coefficient Properties

1. **Symmetry:** For symmetric windows and even derivatives, coefficients are symmetric
2. **Normalization:** Σc_k = 1 for d=0 (constant preservation)
3. **Moment preservation:** Filter preserves polynomials up to degree p

Adaptation for Non-uniform Grids

For non-uniform grids, coefficients are scaled according to local average spacing:

```
// Calculate average spacing in window
h_avg = Σ(x_{i+1} - x_i) / number_of_intervals

// Scaling for d-th derivative
result *= (1/h_avg)^d
```

Efficient Implementation

The program uses LAPACK routine `dposv` for solving the symmetric positive definite system when computing filter coefficients:

```
// Solve linear system for Savitzky-Golay coefficients
dposv_(&uplo, &matrix_size, &nrhs, A, &matrix_size, B, &matrix_size, &info);
```

The coefficient matrix is symmetric and positive definite by construction, ensuring numerical stability.

Modularized Implementation

```
// savgol.h
typedef struct {
    double *y_smooth;     // Smoothed values
    double *y_deriv;      // First derivatives
    int n;                // Number of points
    int poly_degree;      // Polynomial degree
    int window_size;      // Window size
} SavgolResult;

// Coefficient computation
void savgol_coefficients(int nl, int nr, int poly_degree,
                        int deriv_order, double *c);
```

Optimal Properties

The Savitzky-Golay filter minimizes approximation error in the least squares sense and maximizes signal-to-noise ratio for polynomial signals.

Characteristics

Advantages: - Optimal for polynomial signals - Excellent preservation of moments and peak areas - Efficient implementation (convolution) - Minimal phase distortion - Simultaneous computation of functions and derivatives

Disadvantages: - Fixed coefficients for entire window - Problematic for very non-uniform grids - May introduce oscillations at sharp edges - Limited adaptability - Numerical warnings for degrees > 6

Tikhonov Regularization (TIKHONOV)

Theoretical Foundation

Tikhonov regularization solves the ill-posed inverse smoothing problem using a variational approach. We seek a function minimizing the functional:

Continuous formulation:

$$J[u] = (y(x) - u(x))^2 \, dx + (u''(x))^2 \, dx$$

Discrete formulation:

$$J[u] = ||y - u||^2 + ||D^2u||^2$$

where: - $||y - u||^2 = \Sigma(y\_i - u\_i)^2$ is the data fidelity term - $||D^2u||^2 = \Sigma(D^2u\_i)^2$ is the regularization term - is the regularization parameter controlling smoothness - D^2 is the discrete second derivative operator

Second Derivative Discretization

For uniform grid with step h:

$$D^2u\_i \quad (u\_\{i-1\} - 2u\_i + u\_\{i+1\})/h^2$$

For non-uniform grid:

$$D^2u\_i \quad 2[((u\_\{i+1\}-u\_i)/h\_\{i+1\}) - ((u\_i-u\_\{i-1\})/h\_i)]/(h\_i + h\_\{i+1\})$$

Variational Approach

The minimum of functional J[u] satisfies the Euler-Lagrange equation:

$$J/ u\_i = 0 \qquad -2(y\_i - u\_i) + 2 (D^T D u)\_i = 0$$

which leads to the linear system:

$$(I + D^T D)u = y$$

Matrix Representation

For tridiagonal matrix `D²`:

```
       [ 1  -2   1   0  ...  0 ]
D² = 1/h² [ 0   1  -2   1  ...  0 ]
       [                   ]
       [ 0  ...  0   1  -2   1 ]
```

Matrix `A = I +  D^T D` is symmetric, positive definite, banded.

Boundary Conditions

The program implements natural boundary conditions:

```
u''(0) = u''(L) = 0
```

This is realized by matrix modification at edges: - First row: reduced operator - Last row: reduced operator

Adaptive Weights for Non-uniform Grids

Average coefficient method (default):

```
c =   · (1/(n-1)) · Σ(1/h_i²)
```

Local weights method (option -a):

```
w_i =  /h_i²  for each interval
```

Generalized Cross Validation (GCV)

For automatic   selection, we minimize the GCV criterion:

```
GCV( ) = n·RSS( ) / (n - tr(H_ ))²
```

where: - `RSS( ) = ||y - u_ ||²` is the residual sum of squares - `H_  = (I +  D^T D)^{-1}` is the influence matrix - `tr(H_ )` is the matrix trace

Trace estimation using eigenvalues:

```
tr(H_ )   Σ_{k=1}^n 1/(1 +  _k)
```

where ` _k` are eigenvalues of matrix `D^T D`.

Efficient Implementation

The program uses LAPACK routine `dpbsv` for solving symmetric positive definite banded systems:

```
// Banded matrix storage (LAPACK format)
AB[0,j] = superdiagonal elements
AB[1,j] = diagonal elements

// System solution
dpbsv_(&uplo, &n, &kd, &nrhs, AB, &ldab, b, &n, &info);
```

Modularized Implementation

```c
typedef struct {
    double *y_smooth;            // Smoothed values
    double *y_deriv;             // First derivatives
    double lambda;               // Used parameter
    int n;                       // Number of points
    double data_term;            // ||y - u||²
    double regularization_term;  //  ||D²u||²
    double total_functional;     // J[u]
} TikhonovResult;
```

Characteristics

Advantages: - Global optimization with theoretical foundation - Flexible balance between data fidelity and smoothness - Robust to outliers - Efficient for large datasets (O(n) memory and time) - Automatic parameter selection via GCV - Excellent for non-uniform grids

Disadvantages: - Single global parameter - May suppress local details - GCV may fail for some data types - Requires LAPACK library

---

Method Comparison

Computational Complexity

Method	Time	Memory	Scalability
POLYFIT	$O(n \cdot p^3)$	$O(p^2)$	Good for small p
SAVGOL	$O(p^3) + O(n \cdot w)$	$O(w)$	Excellent for large n
TIKHONOV	$O(n)$	$O(n)$	Excellent

Note: w = window size, p = polynomial degree (12), n = number of data points. POLYFIT requires solving a p×p system ($O(p^3)$) for each of n points. SAVGOL solves the p×p system only once for coefficient computation, then applies these coefficients as a linear convolution (O(w)) to all n points. TIKHONOV solves one n×n banded system using efficient LAPACK routines.

Smoothing Quality

Property	POLYFIT	SAVGOL	TIKHONOV
Local adaptability	*****	****	**
Extreme preservation	****	*****	***
Noise robustness	***	****	*****
Derivative quality	*****	*****	***
Boundary behavior	**	***	****
Non-uniform grids	***	**	*****

Practical Recommendations

Method Selection by Data Type

POLYFIT - when:

- Data has variable curvature
- You need to preserve local details
- You have moderately noisy data
- You want highest quality derivatives

SAVGOL - when:

- You want mathematically optimal linear smoothing for polynomial signals
- Data contains periodic or oscillatory components that need preservation
- You need excellent peak shape preservation (areas, moments)
- You want minimal phase distortion in the smoothed signal
- You're processing time series or spectroscopic data
- You need simultaneous high-quality function and derivative estimation
- You have reasonably uniform grid spacing

TIKHONOV - when:

- Data is very noisy
- You need global consistency
- You have significantly non-uniform grid
- You want automatic parameter selection
- You prefer global optimization approaches over local fitting

Parameter Selection

Window size (n) for POLYFIT/SAVGOL:

```
n = 2*k + 1    (odd number)

Recommendations:
- Low noise: n = 5-9
- Medium noise: n = 9-15
- High noise: n = 15-25
```

Polynomial degree (p):

```
- Linear trends: p = 1-2
- Smooth curves: p = 2-3
- Complex signals: p = 3-4
- Advanced applications: p = 5-8
- Maximum: p   12
- Recommended maximum: p < n/2
```

```
Note: Degrees > 6 may cause numerical instability warnings.
```

Lambda ( ) for TIKHONOV:

- Auto selection: `-l auto`
- Manual start: = 0.1
- More smoothing: > 0.1
- Less smoothing: < 0.1
- Range: 10 to 10^3

Usage Examples

Basic Syntax

```
# Polynomial fitting (smoothed values only)
./smooth -m 0 -n 7 -p 2 data.txt

# Polynomial fitting with derivatives
./smooth -m 0 -n 7 -p 2 -d data.txt

# Savitzky-Golay (smoothed values only)
./smooth -m 1 -n 9 -p 3 data.txt

# Savitzky-Golay with derivatives
./smooth -m 1 -n 9 -p 3 -d data.txt

# Tikhonov with automatic   (without derivatives)
./smooth -m 2 -l auto data.txt

# Tikhonov with automatic   and derivatives
./smooth -m 2 -l auto -d data.txt

# Tikhonov with adaptive weights
./smooth -m 2 -l 0.01 -a data.txt

# Tikhonov with adaptive weights and derivatives
./smooth -m 2 -l 0.01 -a -d data.txt
```

Output Format

Without `-d` flag:

```
# Data smooth - aprox. pol. 2dg from 7 points of moving window (least square)
#    x           y
  0.00000E+00  1.00000E+00
  1.00000E+00  2.71828E+00
  ...
```

With `-d` flag:

```
# Data smooth - aprox. pol. 2dg from 7 points of moving window (least square)
#    x           y           y'
  0.00000E+00  1.00000E+00  1.00000E+00
  1.00000E+00  2.71828E+00  2.71828E+00
  ...
```

Working with Non-uniform Grids

```
# Uniformity analysis will be automatic for all methods
./smooth -m 1 -n 7 -p 2 nonuniform_data.txt

# Output may contain:
# Grid uniformity analysis:
#   h_max/h_min = 10.00, CV = 0.450
#   Grid type: NON-UNIFORM
# WARNING: Consider using smaller window size (5) for this non-uniform grid

# For highly non-uniform data use Tikhonov with adaptive weights
./smooth -m 2 -l auto -a highly_nonuniform_data.txt

# With derivatives
./smooth -m 2 -l auto -a -d highly_nonuniform_data.txt
```

Typical Workflow

1. **Quick data exploration:**

   ```
   # Basic smoothing without derivatives
   ./smooth -m 1 data.txt > smooth_data.txt
   ```

2. **Analysis with derivatives:**

   ```
   # Smoothing with derivative computation
   ./smooth -m 1 -d data.txt > smooth_with_deriv.txt
   ```

3. **Optimal smoothing for very noisy data:**

   ```
   # Tikhonov with automatic parameter
   ./smooth -m 2 -l auto data.txt > tikhonov_smooth.txt
   ```

4. **For publication graphics:**

   ```
   # Savitzky-Golay with optimal parameters
   ./smooth -m 1 -n 9 -p 3 -d data.txt > publication_data.txt
   ```

Grid Analysis Module

The `grid_analysis` module provides comprehensive analysis of input data and helps optimize smoothing parameters.

Main Functions

```
// Complete grid analysis
GridAnalysis* analyze_grid(double *x, int n, int store_spacings);

// Quick uniformity check
int is_uniform_grid(double *x, int n, double *h_avg, double tolerance);

// Method recommendation
const char* get_grid_recommendation(GridAnalysis *analysis);

// Optimal window size
int optimal_window_size(GridAnalysis *analysis, int min_window, int max_window);
```

GridAnalysis Structure

```
typedef struct {
    double h_min;            // Minimum spacing
    double h_max;            // Maximum spacing
    double h_avg;            // Average spacing
    double h_std;            // Standard deviation
    double ratio_max_min;    // h_max/h_min ratio
    double cv;               // Coefficient of variation
    double uniformity_score;// Uniformity score (0-1)
    int is_uniform;          // 1 = uniform, 0 = non-uniform
    int n_clusters;          // Number of detected clusters
    int reliability_warning;// Reliability warning
    char warning_msg[512];   // Warning text
} GridAnalysis;
```

Example Analysis Output

```
# Grid uniformity analysis:
#   n = 1000 points
#   h_min = 1.000000e-02, h_max = 1.000000e-01, h_avg = 5.500000e-02
#   h_max/h_min = 10.00, CV = 0.450
#   Grid type: NON-UNIFORM
#   Uniformity score: 0.35
#   Recommendation: High non-uniformity - adaptive methods recommended
```

optimal_window_size Algorithm

The function automatically recommends window size based on grid uniformity:

```
factor = 0.3 + 0.7 * uniformity_score
window = min_window + factor * (max_window - min_window)
```

- **Uniform grid** (score 1.0) $\rightarrow$ larger windows
- **Non-uniform grid** (score < 0.5) $\rightarrow$ smaller windows

---

Compilation and Installation

Requirements

- C compiler (gcc, clang)
- LAPACK and BLAS libraries
- Make (optional)

Compilation using Make

```
# Standard compilation
make

# Clean
make clean

# Install to /usr/local/bin
make install
```

Manual Compilation

```
# Standard compilation
gcc -o smooth smooth.c polyfit.c savgol.c tikhonov.c \
    grid_analysis.c decomment.c -llapack -lblas -lm -O2
```

File Structure

```
smooth/
   smooth.c           # Main program
   polyfit.c/h        # Polynomial fitting module
   savgol.c/h         # Savitzky-Golay module
   tikhonov.c/h       # Tikhonov module
   grid_analysis.c/h  # Grid analysis
   decomment.c/h      # Comment removal
   revision.h         # Program version
   Makefile           # Build system
   README.md          # This documentation
```

Conclusion

The `smooth` program v5.1 provides three complementary smoothing methods in a modular architecture with advanced input data analysis:

- **POLYFIT** - local polynomial approximation using least squares method
- **SAVGOL** - optimal linear filter with pre-computed coefficients
- **TIKHONOV** - global variational method with second derivative regularization
- **GRID_ANALYSIS** - automatic analysis and optimization for non-uniform grids

Each method has a strong mathematical foundation and is optimized for specific data types and applications. Modularization enables easy use of individual methods in other projects. From version 5.1, computation and output of first derivatives is optional via the `-d` flag.

Document revision: 2025-05-31
Compatibility: smooth v5.1+
Dependencies: LAPACK, BLAS