Methods for Smoothing Experimental Data in the smooth Program

Technical Documentation Version 5.5 | November 3, 2025

Contents

Introduction

The `smooth` program implements four 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) + ε_i
```

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

Program Structure

```
./smooth [options] data_file
```

Basic Parameters:

- `-m {0|1|2|3}` - method selection (polyfit|savgol|tikhonov|butterworth)
- `-n N` - smoothing window size (polyfit, savgol)
- `-p P` - polynomial degree (polyfit, savgol, max 12)
- `-l λ` - regularization parameter (tikhonov)
- `-l auto` - automatic λ selection using GCV (tikhonov)
- `-f fc` - normalized cutoff frequency (butterworth, 0 < fc < 0.5)
- `-f auto` - automatic cutoff selection (butterworth, currently returns 0.1)
- `-d` - display first derivative in output (optional, not available for butterworth)
- `-g` - show detailed grid uniformity analysis (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.

Note on grid analysis: The `-g` flag (added in version 5.2) provides detailed grid uniformity statistics helpful for understanding your data and choosing appropriate smoothing parameters.

What's New in Version 5.5

Major Improvements

Butterworth Digital Filter (NEW):

- **4th-order low-pass Butterworth filter** with filtfilt (zero-phase filtering)
- **Low-pass frequency filter:** Removes high-frequency noise, preserves low-frequency trends
- **Cutoff frequency control:** Simple fc parameter controls smoothing strength (lower fc = stronger smoothing)
- **Complex number implementation:** Uses C complex.h for precise pole calculation
- **Scipy-compatible algorithm:** Follows scipy.signal.butter design methodology
- **Filtfilt implementation:** Forward-backward filtering eliminates phase distortion
- **Proper initial conditions:** Implements scipy's lfilter_zi algorithm with companion matrix
- **Edge handling:** Reflection padding minimizes boundary effects
- **Frequency-domain control:** Intuitive cutoff frequency parameter (0 < fc < 0.5)

Previous Version 5.4 Improvements

Tikhonov Regularization (Hybrid Implementation):

- **Automatic discretization selection:** Method automatically chooses between average coefficient (ratio < 2.5) and local spacing (ratio ≥ 2.5) based on grid uniformity
- **Harmonic mean for better accuracy:** Uses harmonic mean for interval averaging in nearly-uniform grids (more accurate than arithmetic mean)
- **Fixed boundary conditions:** Corrected missing superdiagonal element in local spacing method
- **Improved GCV optimization:** Enhanced with over-fitting penalty and L-curve sanity check for large datasets
- **Better functional computation:** Mathematically correct D² discretization matching the matrix formulation

Grid Analysis:

- **Detailed reporting with `-g` flag:** Comprehensive grid uniformity statistics including CV, ratio, spacing details
- **Better recommendations:** Program suggests optimal methods based on detected grid characteristics

---

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)⁰    Σ(x-x_i)¹     ...  Σ(x-x_i)^p  ] [a_0]   [Σy(x-x_i)⁰]
[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
- Works with moderately non-uniform grids

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

CRITICAL: Grid Uniformity Requirement

Version 5.3+ Important Feature: The Savitzky-Golay method now enforces grid uniformity checking.

The mathematical foundation of SG filter assumes **uniformly spaced data points**. The method is based on fitting polynomials in normalized coordinate space where points are at integer positions: {..., -2, -1, 0, 1, 2, ...}.

Uniformity Check:

```
CV = std_dev(spacing) / avg(spacing)

If CV > 0.05:  REJECT - Grid too non-uniform for SG
If CV > 0.01:  WARNING - Nearly uniform, proceed with caution
If CV ≤ 0.01:  OK - Grid sufficiently uniform
```

What happens when grid is rejected:

```
========================================
ERROR: Savitzky-Golay method not suitable for non-uniform grid!
========================================
Grid analysis:
  Coefficient of variation (CV) = 0.2341
  Threshold for uniformity = 0.0500

RECOMMENDED ALTERNATIVES:
  1. Use Tikhonov method: -m 2 -l auto
     (Works correctly with non-uniform grids)
  2. Use Polyfit method: -m 0 -n 5 -p 2
     (Local fitting, less sensitive to spacing)
  3. Resample your data to uniform grid before smoothing
```

The Method of Undetermined Coefficients

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

$$\hat{y}_0 = c_{-nl}\cdot y_{-nl} + \dots + c_0\cdot y_0 + \dots + c_{nr}\cdot y_{nr}$$

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 - but only on uniform grids!

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 = δ_{m,d} · d!    for m = 0,1,...,p
```

where:

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

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);
```

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 **on uniform grids**.

Characteristics

Advantages:

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

Disadvantages:

- **Requires uniform grid** - automatically rejected if CV > 0.05
- Fixed coefficients for entire window
- 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:

$$J[u] = \underbrace{\int(y(x) - u(x))^2\,dx}_{\text{Data fidelity term}} + \lambda\underbrace{\int(u''(x))^2\,dx}_{\text{Smoothness penalty}}$$

Discrete formulation:

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

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** (smoothness penalty)
- λ is the **regularization parameter** controlling the balance
- D^2 is the discrete second derivative operator

The Regularization Parameter λ

The parameter λ is the **heart of Tikhonov regularization** - it controls the balance between fitting the data and smoothing the result.

Physical Interpretation

```
λ = 0:      No smoothing, u = y (exact data fit)
            J[u] = ||y - u||² only

λ → ∞:      Maximum smoothing, u → straight line
            J[u] ≈ λ||D²u||² dominates

λ optimal: Balanced between data fit and smoothness
            Both terms contribute meaningfully
```

Mathematical Role

The minimization of J[u] leads to:

$$(I + \lambda D^TD)u = y$$

Effect of λ on the solution:

- **Small λ (< 0.01):** Matrix ≈ I → solution u ≈ y (minimal smoothing)
- **Large λ (> 1.0):** Matrix ≈ λD^TD → strong curvature penalty (heavy smoothing)

- **Optimal λ:** Matrix components balanced → noise removed, signal preserved

Frequency Domain Interpretation

In Fourier space, Tikhonov acts as a low-pass filter:

```
Ĥ(ω) = 1 / (1 + λω⁴)
```

where ω is spatial frequency.

Effect:

- **Low frequencies (slow variations):** $\hat{H} \approx 1$ → preserved
- **High frequencies (noise, rapid variations):** $\hat{H} \approx 1/(\lambda\omega^4)$ → attenuated
- **Cutoff frequency:** ω_c ∝ λ^(-1/4)

This means:

```
Larger λ  → Lower cutoff → More aggressive low-pass filtering → Smoother result
Smaller λ → Higher cutoff → Less filtering → Result closer to data
```

Practical Guidelines for λ Selection

1. Automatic Selection (RECOMMENDED):

```
./smooth -m 2 -l auto data.txt
```

Uses Generalized Cross Validation (GCV) to find optimal λ.

2. Manual Selection:

Data Characteristics	Recommended λ	Reasoning
Low noise, important details	0.001 - 0.01	Preserve features
Moderate noise	0.01 - 0.1	Balanced (default: 0.1)
High noise	0.1 - 1.0	Strong smoothing
Very noisy, global trends	1.0 - 10.0	Maximum smoothing

3. Iterative Refinement:

```
# Start with automatic
./smooth -m 2 -l auto data.txt

# If result is over-smoothed (details lost):
./smooth -m 2 -l 0.01 data.txt

# If result is under-smoothed (still noisy):
./smooth -m 2 -l 1.0 data.txt
```

4. Diagnostic Criteria:

The program outputs functional components:

```
# Functional J = 1.234e+02 (Data: 5.67e+01 + Regularization: 6.67e+01)
# Data/Total ratio = 0.460, Regularization/Total ratio = 0.540
```

Good balance indicators:

- Data term: 30-70% of total functional
- Regularization term: 30-70% of total functional

Warning signs:

- Data term > 95%: Under-smoothed (λ too small)
- Regularization term > 95%: Over-smoothed (λ too large)

5. Grid-Dependent Considerations:

For non-uniform grids with ratio h_max/h_min > 5:

```
# Start with more conservative (larger) λ
./smooth -m 2 -l 0.5 nonuniform_data.txt

# GCV may be less accurate - check results visually
./smooth -m 2 -l auto nonuniform_data.txt
```

λ and Grid Spacing

The effective regularization strength depends on grid spacing:

```
Effective strength ∝ λ / h²_avg

Same λ on finer grid   → weaker smoothing
Same λ on coarser grid → stronger smoothing
```

For dimensional consistency, λ has units [Length²].

Second Derivative Discretization (Hybrid Method v5.4)

Version 5.4 Implementation: Automatic selection between two discretization schemes based on grid uniformity.

Grid Uniformity Detection

```
ratio = h_max / h_min

ratio < 2.5:  Nearly uniform    → Average Coefficient Method
ratio ≥ 2.5:  Highly non-uniform → Local Spacing Method
```

Method 1: Average Coefficient (for ratio < 2.5)

Used for uniform and mildly non-uniform grids. More robust numerically.

Discretization: For interior point `i` with neighbors at spacing h_left and h_right, use **harmonic mean**:

```
h_harm = 2·h_left·h_right / (h_left + h_right)

D²u_i ≈ (u_{i-1} - 2u_i + u_{i+1}) / h_harm²
```

Why harmonic mean?

- More accurate than arithmetic mean for averaging intervals
- Gives greater weight to smaller spacing (physically correct)
- For h_left = h_right, reduces to standard formula

Matrix construction:

```
c = λ · Σ(1/h_i²) / (n-1)    (average coefficient)

A[i,i]   = 1 + 2c  (interior points)
A[i,i±1] = -c      (off-diagonals)
```

Method 2: Local Spacing (for ratio ≥ 2.5)

Used for highly non-uniform grids. More accurate for variable spacing.

Discretization: For point `i` with left spacing h_1 `= x[i] - x[i-1]` and right spacing h_2 `= x[i+1] - x[i]`:

```
D²u_i ≈ (2/(h₁+h₂)) · [u_{i-1}/h₁ - u_i·(1/h₁+1/h₂) + u_{i+1}/h₂]
```

This is the **correct second derivative formula** for non-uniform grids derived from Taylor expansion.

Matrix construction:

```
w = 2λ / (h₁ + h₂)

A[i,i]   = 1 + w·(1/h₁ + 1/h₂)
A[i,i-1] = -w/h₁
A[i,i+1] = -w/h₂
```

The resulting matrix is:

- Symmetric
- Positive definite
- Tridiagonal (bandwidth = 1)

Boundary Conditions

Natural boundary conditions (second derivative = 0 at ends):

Left boundary (i=0):

```
D²u_0 ≈ (u_1 - u_0) / h_0²

A[0,0] += λ/h_0²
A[0,1] += -λ/h_0²
```

Right boundary (i=n-1):

```
D²u_{n-1} ≈ (u_{n-1} - u_{n-2}) / h_{n-1}²

A[n-1,n-1] += λ/h_{n-1}²
```

Critical fix in v5.4: The boundary superdiagonal element A[0,1] was missing in previous versions, causing isolation of the first point. This is now corrected.

Functional Computation

The actual value of the minimized functional is computed for diagnostic purposes:

Data term:

```
||y - u||² = Σ(y_i - u_i)²
```

Regularization term (must match matrix formulation!):

For **average coefficient method:**

```
||D²u||² = Σ_{interior} [(u_{i-1} - 2u_i + u_{i+1})/h_harm²]²
         + 0.5·[(u_1 - u_0)/h_0²]²
         + 0.5·[(u_{n-1} - u_{n-2})/h_{n-1}²]²
```

For **local spacing method:**

```
||D²u||² = Σ_{interior} [D²u_i]² · (h₁+h₂)/2
         + 0.5·[(u_1 - u_0)/h_0²]² · h_0
         + 0.5·[(u_{n-1} - u_{n-2})/h_{n-1}²]² · h_{n-1}
```

Note the **weighting factors** in local spacing method for proper integration over non-uniform grid.

Total functional:

```
J[u] = ||y - u||² + λ||D²u||²
```

Variational Approach

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

```
∂J/∂u_i = 0  ⟹  -2(y_i - u_i) + 2λ(D^T D u)_i = 0
```

which leads to the linear system:

```
(I + λD^T D)u = y
```

Matrix Representation

Matrix `A = I + λD^T D` is:

- Symmetric
- Positive definite

- Tridiagonal (banded with bandwidth 1)

This structure allows efficient solution using LAPACK's banded solver `dpbsv`.

Generalized Cross Validation (GCV)

For automatic λ selection (`-l auto`), we minimize the GCV criterion:

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

where:

- `RSS(λ) = ||y - u_λ||²` is the residual sum of squares
- `H_λ = (I + λD^T D)^{-1}` is the influence matrix (smoother matrix)
- `tr(H_λ)` is the trace (effective number of parameters)

Interpretation:

- `tr(H_λ)` measures model complexity (degrees of freedom)
- Small λ: tr(H) ≈ n (interpolation, overfitting)
- Large λ: tr(H) ≈ 2 (straight line, underfitting)
- Optimal λ: minimizes prediction error

Trace estimation using eigenvalues:

For uniform grids with natural boundary conditions:

```
tr(H_λ) ≈ Σ_{k=1}^n 1/(1 + λμ_k)

where eigenvalues:
θ_k = πk/n
μ_k = 4·sin²(θ_k/2) / h²
```

Note: This approximation is exact for uniform grids but approximate for non-uniform grids. For highly non-uniform grids (ratio > 5), the program issues a warning.

Enhanced GCV in v5.4

Over-fitting penalty:

```
If tr(H)/n > 0.7:
    GCV_modified = GCV · exp(10·(tr(H)/n - 0.7))
```

This exponential penalty prevents selection of too-small λ that would lead to overfitting.

L-curve backup (for $n > 20000$):

For very large datasets, GCV trace approximation may be inaccurate. The program also computes the L-curve (plot of $||D^2u||^2$ vs $||y-u||^2$) and finds the corner point with maximum curvature:

```
κ = |x'y'' - y'x''| / (x'² + y'²)^(3/2)

where:
x = log(||y - u||²)
y = log(||D²u||²)
```

If GCV and L-curve disagree significantly, the program uses the more conservative (larger) λ.

Efficient Implementation

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

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

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

Complexity:

- Memory: O(n) for banded storage
- Time: O(n) for factorization and back-substitution

This is **optimal** for tridiagonal systems.

Hybrid Implementation (v5.4)

```
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;  // λ||D²u||²
    double total_functional;      // J[u]
} TikhonovResult;

// Main function
TikhonovResult* tikhonov_smooth(double *x, double *y, int n, double lambda);

// Automatic λ selection
double find_optimal_lambda_gcv(double *x, double *y, int n);

// Memory cleanup
void free_tikhonov_result(TikhonovResult *result);
```

Characteristics

Advantages:

- Global optimization with theoretical foundation
- Flexible balance between data fidelity and smoothness (controlled by λ)
- Robust to outliers (quadratic penalty less sensitive than least squares)
- Efficient for large datasets (O(n) memory and time)
- **Automatic λ selection via GCV** - no guessing needed
- **Excellent for non-uniform grids** - correct discretization automatic
- **Unified approach** - same algorithm for uniform and non-uniform grids
- Works well for noisy data with global trends

Disadvantages:

- Single global parameter λ (cannot vary locally)
- May suppress local details if λ too large
- GCV may fail for some data types (especially highly non-uniform grids)
- Requires LAPACK library
- Boundary effects if data has discontinuities at edges

Butterworth Filter (BUTTERWORTH)

Theoretical Foundation

The Butterworth filter is a classical **low-pass frequency filter** in digital signal processing (DSP). It removes high-frequency noise while preserving low-frequency signal trends.

What does "low-pass" mean?

- **Passes low frequencies:** Slow variations in your data pass through unchanged
- **Blocks high frequencies:** Rapid fluctuations (noise) are removed
- **The cutoff frequency (fc)** determines the boundary between "low" and "high"
    - Lower fc → more aggressive smoothing (removes more detail)
    - Higher fc → gentler smoothing (preserves more detail)

The filter is characterized by a **maximally flat magnitude response** in the passband and provides zero phase distortion when implemented as filtfilt.

Filter Transfer Function:

In the analog domain (s-domain), the Butterworth filter has magnitude response:

```
|H(jω)|² = 1 / (1 + (ω/ωc)^(2N))
```

where:

- `N` = filter order (4 in our implementation)
- `ωc` = cutoff frequency (3dB point)
- `ω` = frequency

Key Properties:

- **Maximally flat passband:** No ripples for $\omega < \omega_c$
- **Monotonic rolloff:** Smooth transition from passband to stopband
- **-3dB at cutoff:** $|H(j\omega_c)| = 1/\sqrt{2} \approx 0.707$
- **Rolloff rate:** -20N dB/decade (for N=4: -80 dB/decade)

Digital Implementation

The smooth program implements a **4th-order digital Butterworth low-pass filter** using the following algorithm:

Step 1: Pole Calculation

Butterworth poles lie on unit circle in s-domain at angles:

```
θk = π/2 + π(2k+1)/(2N),  k = 0,1,...,N-1
```

For N=4:

```
s_poles[k] = exp(j·θk)  where θ = {5π/8, 7π/8, 9π/8, 11π/8}
```

Step 2: Frequency Scaling

Scale poles by prewarped cutoff frequency:

```
wc = tan(π·fc)     (prewarp for bilinear transform)
s_poles_scaled = wc · s_poles
```

Step 3: Bilinear Transform

Convert analog poles to digital domain:

```
z_poles = (2 + s_poles_scaled) / (2 - s_poles_scaled)
```

The bilinear transformation maps:

- Left half of s-plane → inside unit circle in z-plane
- jω axis → unit circle in z-plane
- Preserves stability

Step 4: Biquad Cascade

Form two 2nd-order sections (biquads) from conjugate pole pairs:

```
H(z) = H1(z) · H2(z)

Each biquad: H_i(z) = (b0 + b1·z⁻¹ + b2·z⁻²) / (1 + a1·z⁻¹ + a2·z⁻²)
```

This approach provides better numerical stability than direct 4th-order implementation.

Filtfilt Algorithm

The **filtfilt** (forward-backward filtering) eliminates phase distortion:

Algorithm:

1. **Pad signal:** Reflect signal at boundaries (3×order length)
2. **Forward filter:** Apply H(z) from left to right → y_fwd
3. **Reverse:** y_rev = reverse(y_fwd)
4. **Backward filter:** Apply H(z) to y_rev → y_bwd
5. **Reverse back:** y_final = reverse(y_bwd)
6. **Extract:** Remove padding to get final result

Effect:

- **Zero phase lag:** No signal delay
- **Effective order:** 2N = 8 (squared magnitude response)
- **Steeper rolloff:** $|H\_eff(j\omega)|^2 = |H(j\omega)|^4$

Initial Conditions (lfilter_zi)

To minimize edge transients, we compute initial filter state using **scipy's lfilter_zi algorithm**:

Problem: Find initial state `zi` such that for constant input `x = c`:

```
zi = A·zi + B·c
```

This ensures the filter starts in steady-state, eliminating startup transients.

Solution: Solve linear system using companion matrix:

```
(I - A)·zi = B

where:
  A = companion(a).T    (companion matrix of denominator)
  B = b[1:] - a[1:]·b[0]
```

The companion matrix for `[1, a1, a2, a3, a4]` is:

```
     [a1  -a2  a3  -a4]
     [1     0   0    0 ]
     [0     1   0    0 ]
     [0     0   1    0 ]
```

Normalized Cutoff Frequency

The cutoff frequency `fc` is **normalized** to the sampling rate and is the **most important parameter** for Butterworth filtering.

Simple explanation:

- `fc` controls how much smoothing you get
- **Smaller fc (e.g., 0.05)** → heavy smoothing, only very slow trends preserved
- **Larger fc (e.g., 0.30)** → light smoothing, more detail preserved
- Valid range: `0 < fc < 0.5` (Nyquist limit)

Technical details:

```
fc = f_cutoff / f_sample


where:
  f_cutoff = desired cutoff frequency in physical units
  f_sample = 1 / h_avg  (h_avg = average data spacing)
```

Nyquist Constraint: `0 < fc < 0.5`

- `fc = 0.5` corresponds to Nyquist frequency (f_sample/2) - maximum possible
- Higher fc → less filtering (more high frequencies pass)
- Lower fc → more filtering (smoother result)

Physical Interpretation:

Example: Data with spacing h_avg = 0.1 seconds

- Sample rate: f_sample = 1/0.1 = 10 Hz
- Nyquist frequency: 5 Hz
- If fc = 0.2, then f_cutoff = 0.2 × 10 = 2 Hz
- Filter removes frequencies above ~2 Hz

Practical Guidelines for Choosing fc:

fc Value	Smoothing Strength	When to Use
0.01 - 0.05	**Very strong**	Extremely noisy data, only global trends matter
0.05 - 0.15	**Moderate**	Typical experimental data with noise
0.15 - 0.30	**Light**	Good quality data, preserve features
> 0.30	**Minimal**	Low noise, want to keep almost everything

Quick Start Recommendations:

- **Not sure? Start with fc = 0.15** - good balance for most data
- **Too noisy after smoothing?** Decrease fc (e.g., try 0.10)
- **Lost important details?** Increase fc (e.g., try 0.25)
- **Extreme noise?** Try fc = 0.05
- **High quality data?** Try fc = 0.25 - 0.30

IIR Filter Implementation

Uses **Transposed Direct Form II** for numerical stability:

```
For each sample n:
  y[n] = b[0]·x[n] + z[0]
  z[0] = b[1]·x[n] - a[1]·y[n] + z[1]
  z[1] = b[2]·x[n] - a[2]·y[n] + z[2]
  z[2] = b[3]·x[n] - a[3]·y[n] + z[3]
  z[3] = b[4]·x[n] - a[4]·y[n]
```

where `z[]` is the filter state (4 elements for 4th order).

Modularized Implementation

```
// butterworth.h
typedef struct {
    double *y_smooth;     // Smoothed values
    int n;                // Number of points
    int order;            // Filter order (4)
    double cutoff_freq;   // Normalized cutoff frequency
    double sample_rate;   // Effective sample rate (1/h_avg)
} ButterworthResult;

// Main function
ButterworthResult* butterworth_filtfilt(double *x, double *y, int n,
                                        double cutoff_freq, int auto_cutoff);

// Automatic cutoff selection (currently returns 0.1)
double estimate_cutoff_frequency(double *x, double *y, int n);

// Memory cleanup
void free_butterworth_result(ButterworthResult *result);
```

Grid Requirements

IMPORTANT: Butterworth filter works best with **uniform or nearly-uniform grids**.

The filter assumes uniform sampling when computing the cutoff frequency. For highly non-uniform grids:

```
Grid Uniformity (ratio = h_max/h_min):
  ratio < 5:   Good - Butterworth works well
  5 ≤ ratio < 20: Acceptable - may have suboptimal performance
  ratio ≥ 20:  Rejected - use Tikhonov instead
```

Why uniform grids?

- Frequency analysis assumes constant sampling rate
- fc is defined relative to sample rate
- Non-uniform sampling distorts frequency response

For non-uniform grids: Use Tikhonov method (`-m 2 -l auto`) which handles arbitrary spacing correctly.

Characteristics

Advantages:

- **Zero phase distortion** (filtfilt eliminates all phase lag)
- **Maximally flat frequency response** in passband
- **Classical DSP approach** with extensive literature and understanding
- **Predictable frequency-domain behavior** - easy to interpret cutoff frequency
- **No ringing** (unlike Chebyshev or elliptic filters)
- **Efficient implementation** - O(n) time complexity
- **Smooth monotonic rolloff** - natural attenuation curve

Disadvantages:

- **Requires uniform/nearly-uniform grid** (ratio < 20)
- **No derivative output** (Butterworth is smoothing-only)
- **Less local adaptability** than polynomial methods
- **Cutoff selection not automatic** (currently manual tuning needed)
- **Edge effects** despite padding
- **Frequency interpretation** may be less intuitive than λ for some users

Comparison with Other Methods

BUTTERWORTH vs SAVITZKY-GOLAY:

- Both assume uniform grids
- **Butterworth:** True frequency-domain filtering, maximally flat passband
- **Savitzky-Golay:** Polynomial approximation in time domain
- **Choose Butterworth for:** Periodic signals, spectral data, frequency-domain interpretation
- **Choose Savitzky-Golay for:** Polynomial trends, peak detection, derivative estimation

BUTTERWORTH vs TIKHONOV:

- **Butterworth:** Classical signal processing, frequency-domain control
- **Tikhonov:** Variational optimization, works with non-uniform grids
- **Choose Butterworth for:** Uniform data, need frequency-domain understanding
- **Choose Tikhonov for:** Non-uniform grids, mathematical optimization approach

BUTTERWORTH vs POLYFIT:

- **Butterworth:** Global frequency filtering, uniform smoothing

- **Polyfit:** Local polynomial fitting, adapts to curvature changes
- **Choose Butterworth for:** Stationary signals, spectroscopic data
- **Choose Polyfit for:** Variable curvature, local feature preservation

---

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
BUTTERWORTH	$O(n)$	$O(n)$	Excellent

Note: w = window size, p = polynomial degree (≤12), n = number of data points.

Smoothing Quality

Property	POLYFIT	SAVGOL	TIKHONOV	BUTTERWORTH
Local adaptability	*****	****	**	**
Extreme preservation	****	*****	***	***
Noise robustness	***	****	*****	*****
Derivative quality	*****	*****	***	N/A
Boundary behavior	**	***	****	***
Non-uniform grids	***	✗	*****	**
Ease of use (v5.5)	****	****	*****	****
Parameter selection	Manual	Manual	Auto (GCV)	Manual
Frequency control	No	No	No	Yes
Phase distortion	N/A	N/A	N/A	Zero

Key: ✗ = Not suitable (automatically rejected)

Grid Type Compatibility

Grid Type	POLYFIT	SAVGOL	TIKHONOV	BUTTERWORTH
Perfectly uniform (CV < 0.01)	✓	✓	✓	✓
Nearly uniform (CV < 0.05)	✓	⚠	✓	✓
Moderately non-uniform (0.05 < CV < 0.2)	✓	✗	✓	⚠
Highly non-uniform (CV > 0.2)	⚠	✗	✓	⚠

Legend:

- ✓ = Recommended
- ⚠ = Usable with caution

- ✗ = Rejected or not recommended
- * = Uses local spacing method automatically

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
- Grid has moderate spacing variations
- You need local adaptability

SAVGOL - when:

- **Grid is uniform (CV < 0.05)** - automatically checked!
- 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 on uniform grids
- You need simultaneous high-quality function and derivative estimation
- Computational efficiency is critical (large datasets)

TIKHONOV - when:

- **Grid is non-uniform** - works perfectly automatically!
- Data is very noisy
- You need global consistency
- **You want automatic parameter selection (λ auto)** - highly recommended!
- You prefer global optimization approaches over local fitting
- You want robust handling of outliers
- You want the simplest workflow (one parameter, automatic selection)
- You need to process very large datasets efficiently
- **You're not sure which method to use** - Tikhonov with `-l auto` is safest!

BUTTERWORTH - when:

- **Grid is uniform or nearly-uniform (ratio < 20)** - essential requirement!
- You want to **remove high-frequency noise** while keeping slow trends
- You need **simple frequency-based smoothing** - just set cutoff frequency fc

- You want **zero phase distortion** (no signal delay)
- Data is periodic, oscillatory, or spectroscopic
- You need **frequency-domain interpretation** of filtering
- Data is from instrumentation with known sampling rate
- You need **predictable frequency response** (maximally flat passband)
- Working with time-series data at constant sampling
- You understand or want to learn about cutoff frequency concept

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

Rule of thumb: n ≈ 2p + 3
```

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 (λ) for TIKHONOV:

```
**RECOMMENDED:**
- Auto selection: -l auto  (uses GCV optimization)

**MANUAL SELECTION:**
Starting points by noise level:
- Low noise:        λ = 0.001 - 0.01
- Medium noise:     λ = 0.01 - 0.1   (default: 0.1)
- High noise:       λ = 0.1 - 1.0
- Very noisy data:  λ = 1.0 - 10.0

Full range: 10⁻⁶ to 10³

**ITERATIVE REFINEMENT:**
1. Start with -l auto
2. Check functional balance (should be 30-70% each)
3. If over-smoothed: decrease λ by factor of 10
4. If under-smoothed: increase λ by factor of 10
5. Repeat until satisfied

**GRID-DEPENDENT:**
For non-uniform grids (ratio > 5):
- Start more conservative (larger λ)
- GCV may be less accurate - check visually
```

Cutoff frequency (fc) for BUTTERWORTH:

```
**RECOMMENDED:**
Start with manual selection: fc = 0.15 - 0.20

**MANUAL SELECTION by noise level:**
- Low noise:        fc = 0.20 - 0.30  (preserve details)
- Medium noise:     fc = 0.15 - 0.20  (typical, recommended)
- High noise:       fc = 0.05 - 0.15  (aggressive smoothing)
- Very noisy data:  fc = 0.01 - 0.05  (heavy smoothing)

Full range: 0 < fc < 0.5 (Nyquist limit)

**AUTOMATIC SELECTION:**
- Use -f auto (currently returns default fc = 0.1)
- Note: Automatic selection not yet fully implemented
- Manual tuning recommended for best results

**PHYSICAL INTERPRETATION:**
fc = f_cutoff / f_sample
where f_sample = 1 / h_avg

Example: h_avg = 0.1 sec → f_sample = 10 Hz
         fc = 0.2 → f_cutoff = 2 Hz (removes freq > 2 Hz)

**ITERATIVE REFINEMENT:**
1. Start with fc = 0.15 or fc = 0.20
2. If result too smooth (details lost): increase fc
3. If result too noisy (not smooth enough): decrease fc
4. Typical adjustment: ±0.05
5. Repeat until satisfied

**GRID-DEPENDENT:**
For non-uniform grids (ratio > 5):
- Results may be suboptimal
- Consider using Tikhonov instead
- If ratio > 20: automatically rejected
```

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)
# NOTE: Will be rejected if grid is non-uniform!
./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 λ (RECOMMENDED)
./smooth -m 2 -l auto data.txt

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

# Tikhonov with manual λ
./smooth -m 2 -l 0.01 data.txt

# Tikhonov with manual λ and derivatives
./smooth -m 2 -l 0.01 -d data.txt

# Butterworth with manual cutoff frequency
./smooth -m 3 -f 0.15 data.txt

# Butterworth with automatic cutoff (currently returns 0.1)
./smooth -m 3 -f auto data.txt

# Grid analysis (works with any method)
./smooth -m 2 -l auto -g data.txt
```

Output Format

Without `-d` flag:

```
# Data smooth - Tikhonov regularization with lambda = 1e-01
# Functional J = 1.234e+02 (Data: 5.67e+01 + Regularization: 6.67e+01)
# Data/Total ratio = 0.460, Regularization/Total ratio = 0.540
#    x          y
  0.00000E+00  1.00000E+00
  1.00000E+00  2.71828E+00
  ...
```

With `-d` flag:

```
# Data smooth - Tikhonov regularization with lambda = 1e-01
# Functional J = 1.234e+02 (Data: 5.67e+01 + Regularization: 6.67e+01)
# Data/Total ratio = 0.460, Regularization/Total ratio = 0.540
#    x          y          y'
  0.00000E+00  1.00000E+00  1.00000E+00
  1.00000E+00  2.71828E+00  2.71828E+00
  ...
```

With `-g` flag (grid analysis):

```
# ========================================
# GRID UNIFORMITY ANALYSIS
# ========================================
# Grid uniformity analysis:
#   n = 1000 points
#   h_min = 9.500000e-03, h_max = 1.200000e-02, h_avg = 1.000000e-02
#   h_max/h_min = 1.26, CV = 0.052
#   Grid type: NON-UNIFORM
#   Uniformity score: 0.94
#   Standard deviation: 5.200000e-04
#   Detected clusters: 0
#   Recommendation: Grid is nearly uniform - standard methods work well
# ========================================
#
# Using Average Coefficient Method (h_max/h_min = 1.26)
# GCV optimization for n=1000 points (h_max/h_min = 1.26)
...
```

Working with Non-uniform Grids (v5.4)

```
# First, analyze your grid
./smooth -m 2 -l auto -g nonuniform_data.txt

# If ratio < 2.5, program uses average coefficient method
# Output: "Using Average Coefficient Method (h_max/h_min = 1.85)"

# If ratio ≥ 2.5, program uses local spacing method
# Output: "Using Local Spacing Method (h_max/h_min = 8.42)"

# Automatic selection - no user intervention needed!
# Just use the same command:
./smooth -m 2 -l auto nonuniform_data.txt

# For very non-uniform grids (ratio > 5), you may see:
# "WARNING: Highly non-uniform grid detected!"
# "GCV trace approximation may be less accurate."
# In this case, try manual λ or check results visually.
```

Typical Workflow

1. **Quick data exploration with grid analysis:**

```
# Check grid uniformity
./smooth -m 2 -l auto -g data.txt > smooth_data.txt

# Review output comments for:
# - Grid uniformity (CV, ratio)
# - Which discretization method was used
# - Functional balance (data vs regularization)
```

2. **Choose method based on grid:**

```
# For uniform grids (CV < 0.05):
./smooth -m 1 -n 9 -p 3 -d data.txt

# For non-uniform grids:
./smooth -m 2 -l auto -d data.txt
```

3. **Refine λ if needed:**

```
# If automatic λ gives over-smoothing:
./smooth -m 2 -l 0.01 -d data.txt

# If under-smoothing:
./smooth -m 2 -l 1.0 -d data.txt
```

4. **For publication graphics:**

```
# Final smoothing with derivatives
./smooth -m 2 -l auto -d data.txt > publication_data.txt

# Check functional balance in output comments
# Ideal: both terms contribute 30-70%
```

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

```c
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
    double *spacings;        // Array of spacings (optional)
    int n_points;            // Number of points
    int n_intervals;         // Number of intervals (n-1)
} GridAnalysis;
```

Example Analysis Output

```
# ========================================
# GRID UNIFORMITY ANALYSIS
# ========================================
# 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
#   Standard deviation: 2.475000e-02
#   Detected clusters: 2
#   Recommendation: High non-uniformity - adaptive methods recommended
# WARNING: HIGH grid non-uniformity: h_max/h_min = 10.0
# Adaptive methods are strongly recommended.
# Consider using smaller regularization parameters.
#
# WARNING: 2 abrupt spacing changes detected (possible data clustering).
# Standard methods may over-smooth clustered regions.
# ========================================
```

Grid Uniformity Thresholds

```
CV < 0.01:   Perfectly uniform - all methods work optimally
             → POLYFIT, SAVGOL, TIKHONOV all excellent

CV < 0.05:   Nearly uniform - SAVGOL works with warning
             → SAVGOL may show warning but works
             → POLYFIT and TIKHONOV work fine

0.05 ≤ CV < 0.20: Moderately non-uniform
             → SAVGOL rejected automatically
             → POLYFIT usable
             → TIKHONOV recommended (uses average coef if ratio < 2.5)

CV ≥ 0.20:   Highly non-uniform
             → SAVGOL rejected
             → POLYFIT with caution
             → TIKHONOV strongly recommended (may use local spacing)

RATIO:
ratio < 2.5:  Tikhonov uses average coefficient method
ratio ≥ 2.5:  Tikhonov uses local spacing method
ratio > 10:   Warning issued, GCV may be less accurate
```

Compilation and Installation

Requirements

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

Compilation using Make

```
# Standard compilation
make

# Debug build
make debug

# Clean
make clean

# Install to user's home directory
make install-user

# Install to system (requires root)
make install
```

Manual Compilation

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

# With warnings
gcc -Wall -Wextra -pedantic -o smooth smooth.c polyfit.c savgol.c \
    tikhonov.c butterworth.c grid_analysis.c decomment.c -llapack -lblas -lm -O2
```

File Structure

```
smooth/
├── smooth.c            # Main program (v5.5: added Butterworth)
├── polyfit.c/h         # Polynomial fitting module
├── savgol.c/h          # Savitzky-Golay module (v5.3: with uniformity check)
├── tikhonov.c/h        # Tikhonov module (v5.4: hybrid implementation)
├── butterworth.c/h     # Butterworth filter module (v5.5: new)
├── 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 v5.5 provides four 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 (uniform grids only)
- **TIKHONOV** - global variational method with hybrid automatic discretization
- **BUTTERWORTH** - digital low-pass filter with zero-phase filtfilt (NEW in v5.5)
- **GRID_ANALYSIS** - automatic analysis and method recommendation

Version 5.5 Highlights

New Addition:

- **Butterworth low-pass filter (4th-order)** - classical DSP frequency filter with filtfilt for zero-phase smoothing
  - Removes high-frequency noise while preserving low-frequency trends
  - Simple cutoff frequency parameter fc controls smoothing strength
  - Complex number implementation for precise pole calculation
  - Scipy-compatible algorithm (follows scipy.signal.butter)
  - Maximally flat frequency response in passband
  - Proper initial conditions via lfilter_zi algorithm
  - Frequency-domain control with normalized cutoff parameter

Previous Version 5.4 Improvements

1. **Tikhonov hybrid implementation** - automatic selection between average coefficient (ratio < 2.5) and local spacing (ratio ≥ 2.5) methods
2. **Harmonic mean** - more accurate interval averaging for nearly-uniform grids
3. **Fixed boundary conditions** - corrected missing superdiagonal element
4. **Enhanced GCV** - over-fitting penalty and L-curve backup for large datasets
5. **Better diagnostics** - functional balance reporting, λ selection guidance
6. **Grid analysis with `-g`** - detailed uniformity statistics for parameter optimization

When to Use Each Method

Quick Decision Tree:

```
Is your grid uniform (CV < 0.05)?
├─ YES: Multiple good options:
│   ├─ SAVGOL: Best for polynomial signals with derivatives
│   │   smooth -m 1 -n 9 -p 3 -d data.txt
│   ├─ BUTTERWORTH: Best for frequency-domain interpretation
│   │   smooth -m 3 -f 0.15 data.txt
│   └─ TIKHONOV: Universal choice with auto parameters
│       smooth -m 2 -l auto -d data.txt
│
└─ NO (non-uniform): Use TIKHONOV for correct handling
    smooth -m 2 -l auto -d data.txt

Need frequency-domain control?
└─ Use BUTTERWORTH (requires uniform grid)
    smooth -m 3 -f 0.15 data.txt

Need local adaptability?
└─ Use POLYFIT regardless of grid
    smooth -m 0 -n 7 -p 2 -d data.txt

Not sure?
└─ Use TIKHONOV with automatic λ - safest choice!
    smooth -m 2 -l auto -g -d data.txt
```

Each method has a strong mathematical foundation and is optimized for specific data types. The program provides automatic guidance on method selection and parameters, with extensive diagnostics to ensure correct usage.

Best Practices

1. **Always check grid first:** Use `-g` flag to understand your data
2. **Start with automatic:** Use `-l auto` for Tikhonov, let GCV find optimal λ
3. **Check functional balance:** Look for 30-70% split between data and regularization terms
4. **Iterate if needed:** Adjust λ manually if automatic selection doesn't satisfy requirements
5. **Use derivatives wisely:** Add `-d` only when needed - cleaner output without it
6. **Understand the trade-off:** More smoothing (larger λ) = more noise reduction but less detail

---

Document revision: 2025-10-13

Program version: smooth v5.4

Dependencies: LAPACK, BLAS

License: See source files